続 新しいプログラミング・パラダイム

井田哲雄・田中二郎 編



共立出版株式会社

執筆者一覧（五十音順）

井田	哲雄	筑波大学 電子・情報工学系
上田	和紀	(財)新世代コンピュータ技術開発機構
桜川	貴司	京都大学 数理解析研究所
田中	二郎	富士通(株) 国際情報社会科学研究所
富樫	敦	東北大学 電気通信研究所
外山	芳人	NTT 基礎研究所
中川	中	(株)SRA ソフトウェア工学研究所
林	晋	龍谷大学 理工学部 数理情報学科
二木	厚吉	電子技術総合研究所 言語システム研究室
横内	寛文	日本アイ・ビー・エム(株) 東京基礎研究所
横田	一正	(財)新世代コンピュータ技術開発機構

まえがき

「プログラミング・パラダイム」という言葉が，コンピュータ・サイエンスの研究者の間で市民権を得てから久しい．プログラミング・パラダイムとは，プログラムを書く上での，考え方の指針を与えるものである．いわば，プログラムの文体と，プログラミング言語の中間に位置する概念である．「プログラムの文体」というと，プログラミングに対する個人の美意識が作る形式といった趣きがあるが，「プログラミング・パラダイム」といったときには，もっと客観的な「プログラムを作り出すときの，与えられた問題のモデル化の規範」を示していると考えられる．このプログラミング・パラダイムは，それが定着すると，やがて具体的な「プログラミング言語」へと結実していくと考えられる．

では，なぜ，今パラダイムを問題としなければならないのだろうか．それには，少しばかりプログラミングの歴史を振り返ってみなければならない．はじめてコンピュータが作られ，機械語やアセンブリ言語でプログラムが書かれていた時代には，プログラミング・パラダイムという概念は希薄であった．しかし，Fortran, Cobol といったいわゆる高水準言語が1950年代の終りに開発されるようになると，プログラミング・パラダイムという概念の萌芽が見られるようになる．Fortran や Cobol などでは，これらの言語で書かれたプログラムが実行されるフォン・ノイマン型のコンピュータ・アーキテクチャが強く意識されている．ここでのプログラミング・パラダイムとは，コンピュータへの演算実行命令をひとまとめにした「手続き型」のパラダイムであった．

この手続き型パラダイムは，1960年代の後半に提示された E. W. Dijkstra (E. W. ダイクストラ) の「go to 文有害説」により，一つの曲り角を迎える．この説は「go to 文はプログラムの構造を不用意に複雑にし，プログラムの性質の理解を著しく困難にするから，有害である」と主張するものである．これを契機として，コンピュータ・サイエンスの研究者の間に，プログラムの構造や理論についての関心が高まり，構造的プログラミングなどのプログラム設計論が生まれる．またそれと同時に，新たなパラダイムを反映させた Pascal のようなプログラミング言語も作られるようになる．

1970年代に入ると，プログラミング・パラダイムにとって，二つ目の曲り角がやってくる．それは，フォン・ノイマン型コンピュータを前提としない新しい計算の方法論の提案に伴って起きたものである．ここでの新しいパラダイムは，「関数型」，「論理型」，「オブジェクト指向型」パラダイムで代表される．人間の思考形態により近づくことをめざしたものである．これらの第二世代のプログラミング・パラダイムに基づき，FP, Prolog, Smalltalk などのさまざまなプログラミング言語が提案され，また既存のプログラミング言語にもさまざまな影響を与えた．

これらの新しいパラダイムは，提案されたままの形で固定化するのではなく，研究者の間にさらに新たな刺激や知的挑戦を与えた．パラダイムの進歩は，とどまることなく，より計算の本質に迫るような新しいパラダイムが1980年代になってからつぎつぎに登場する．そしてパラダイム戦国時代の様相を示しだす．それはプログラミング・パラダイムにとって三つ目の曲り角であるということができよう．本書は，こうした，第三世代のプログラミング・パラダイムについて解説を行なったものである．それぞれのパラダイムの理念を整理して読者に語りかけたものである．本書で示したパラダイムを分類するのは，時期尚早と思われる．しかしながら，本書に挙げたパラダイムから，ある共通のすなわち方向性，『関数や代数などの数学的概念との接近』，『論理への接近』，『並列・分散への対処』など，幾つかの傾向を多くの読者は読みとることであろう．

本書は，共立出版（株）のコンピュータ・サイエンス誌 **bit** に1987年11月号から1989年4月号にわたって掲載された連載記事を収録したものである．すで

に連載記事のほぼ半分については，1989年11月に単行本『新しいコンピュータ・パラダイム』として出版されている．本書はその残り半分を収録した続編に相当する．既出版のほうには，『新しいプログラミング・パラダイムをめぐって』，『ラムダ計算と高階プログラミング』，『マルセイユ Prolog, Prolog II, Prolog III』，『制約論理型プログラミング』，『オブジェクト指向』，『型推論とML』，『Miranda』，『項書換えシステムと完備化手続き』，『等式プログラミングから融合型プログラミングへ』を収録している．

本書は，前半の執筆の精神を継承しつつ，内容的には独立した九つの解説を収録している．既出版のほうも合わせ読まれると，さらにプログラミング・パラダイムの理解を豊富にすることができよう．

なお，単行本化にあたっては，あらたに加筆，修正を行ない，また，引用文献の更新，用語の統一なども行なって，本全体として読みやすくなるように配慮した．

本書は，コンピュータ科学を専攻する大学生，大学院生，あるいはコンピュータ関連の業務に従事する読者を想定して書かれている．本書がこれらの方々に，知的インスピレーションを提供することができるならば幸いである．

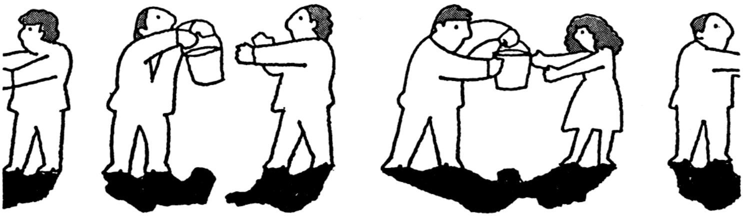
1990年10月

井田哲雄
田中二郎

目次

第1章　並行プログラミングとGHC

上田和紀



1.1　はじめに

新しいプログラミング・パラダイムというと，読者の方々はどのようなキーワードを思い浮かべるだろうか？　関数型，論理型，オブジェクト指向などをまず思い浮かべるのではないかと思う．どういうわけか，本章の主題である**並行プログラミング**は，パラダイムとして市民権を得ているとはいい難い．本章の目的は，この並行プログラミングが，実はごくふつうのプログラミングのためのパラダイムとして，興味深くしかも重要なものであることを示すことである．

説明に用いる並行プログラミング言語としては，GHC[11),37),38)] を採用したい．GHC は，論理プログラミングにおける並列性の研究から 1984 年末に筆者が提案した言語であり，ICOT（(財)新世代コンピュータ技術開発機構）の並列推論マシンの核言語（KL1）の基本ともなっている言語である．GHC を使うのは，徹底的に並列であり，また非常に単純だからである．GHC は並列論理型言語と呼ばれることが多いが，ここでの話に関する限り，論理とのかかわりを考える必要はない．

注目に値する並行プログラミング言語には，ほかにも Qlisp[13)] や Multilisp[15),16)] などの並行 Lisp[24)]，ABCL/1[34)] などの並行オブジェクト指向言語[41)]，Linda[1)] のようにほかの逐次プログラミング言語に並行プログラミング機能を付加する付加型言語などがあって，目移りするかもしれない．だが，これらの言語を本質

的によく理解するためには，GHCのような単純で原始的な言語をまず理解しておくことは有用であろう．

1.2　ターゲットを明確にしよう

まず最初に，**並行**という用語は，計算機科学において**並列**という用語とどのように区別されて用いられるのかを説明しておこう．実をいうと，この二つの用語はいつも厳密に使い分けられている，あるいは使い分けることができるわけではないのだが，使い分けるときには，以下のような区別になっている．

並行というのはconcurrentの訳語で，複数の仕事（それぞれの仕事は，より細かい仕事の組合せからなる大きなものでもよい）が時間的にオーバラップして進行する，つまりある仕事が，ほかの仕事の終了を待たずに始まることをいう．このとき，一つの処理主体（人ないし計算機）がすべての仕事を受け持って，それぞれの仕事を少しずつ進めるのか，それとも複数の処理主体が同時に分担して行なうのかは問題ではない．一方，並列というのはparallelの訳語で，複数の仕事が，複数の処理主体によって同時に進むことをいう．別の言い方をすると，並行というのは，論理的・巨視的な同時性，並列というのは物理的・微視的な同時性を指しているといってもよい．

プログラミングというのは，第一義的には**何を**計算するかを指定するものであり，それを実際に**どう**処理するかは，かなり処理系の裁量に任されているといってよい．並行プログラミングは，この第一義的な部分を記述するための一つのパラダイムと位置づけることができる．では具体的に，パラダイムとしての並行プログラミングは，どのようなものを考えればよいだろうか．以下に，筆者なりの考えをまとめてみよう．

① 通常**プロセス**と呼ばれる，並行して実行される構成要素と，それらの間の相互作用を記述することによって，プログラムを構成するもの．かりにFortranで書いた数値計算プログラムや，Prologで書いた探索プログラムが，結果として並列計算機の上で並列に実行できるものであったとしても，それだけでは並行プログラミングをしたことにはならない．何と何を並行して行なうかを，プログラミングの上で意識し，それをプロ

グラム上に反映させることが，プログラミング・パラダイムとしては重要である．

② “ふつう”のプログラムを対象にして，積極的に行なうもの．“ふつうの”とは，逐次プログラミングでも書ける，という程度の意味である．これまでは，並行プログラミングというと，オペレーティング・システムのように，プロセスの考え方がないとプログラミングが非常に困難になるような場面で，必要に迫られて行なうものであった．が，そのような限定された場面でしか並行プログラミングを考えないのでは，技術としても発展しないし，並行ソフトウェアの蓄積も図れない．

③ Programming in the large，つまりプログラムをその部品（プロセス）から構成する方法論を与えるもの．従来は，並行プログラミングというと，programming in the small，つまり言語に備わっている基本操作を用いて並行プログラムの断片を記述する方法に，興味の中心が集まることが多かった．これももちろん重要だが，それだけ考えていたのでは，トイ・プログラミングでない，本格的なプログラミングのためのパラダイムとしての並行プログラミングが発展しない．

④ 望むらくは，並列計算機の性能（つまり並列性）を活かすための方法論を与えるもの．プログラミングにおいて並行処理の概念を活用することと，プログラムを並列に実行することとは，実は独立であって[36]，並行プログラミング言語で書いたからといって，並列計算機の上で効率良く実行できるとは限らない．本質的に並列処理に向かない問題ならば，これはやむを得ないことである．逆に，前述の Fortran の例のように，逐次プログラムからベクトル処理などの形で並列性を取り出すこともある．

だが，並列計算機の性能を活かす方法を考えるのに，思考の道具として並行プログラミング言語を持つことはやはり大切で，その上で並列実行の観点からより優れたプログラムの研究開発を進めてゆくのが本道であろう．また，それが，並行プログラミングに期待される大きな役割であることも確かである．

パラダイムとしての並行プログラミングは，オブジェクト指向[23),42)]と似た

ところを持っている．つまりこれらは，ある程度言語に独立な概念であって，プロセスなりオブジェクトなりを記述できる言語さえあれば，あとは言語機能よりもむしろ実践のほうが問題にされる．パラダイムの根幹を支える安定した理論体系の整備が待たれる，という点でも似ている．もっと技術的にみても，並行プログラミングとオブジェクト指向は非常に近い．プロセスとは，並行して動作し，互いに通信しあうオブジェクトにほかならないからである．

1.3　はじめが大切

並行プログラミングを勉強するのに，一体何から始めるのがよいだろうか？1970年代の後半まで，この分野の代表的文献といえば，Dijkstra の Co-operating Sequential Processes という論文[6]であった．メモリを共有する手続き型プロセス間の同期の問題を扱った古典的な論文である．手元にある Filman と Friedman の教科書の訳書[10]を見ると，「この論文は，並行処理プログラムがいかに難しいものであるかを明らかにした優れた論文である」(p. 47) とある．だが，別の見方をすると，この論文は，「並行プログラミングというものはとにかく難しいものである」という意識を多くの人々に植え付けてしまったのではないかと思う．並行プログラミングを研究として志す人にとって，今でも一度は読むべき文献であることに変わりはないが，初心者にはもっと抽象度の高いものから与えなければ，よくて消化不良，ふつうは拒絶反応を起こしてしまう．

一方，1978年に出版された Hoare の Communicating Sequential Processes (CSP) に関する論文[19]は，Dijkstra の論文と比べるとはるかに簡明に，並行プログラミングの面白さを示したものだった．

CSP のわかりやすさは，次の2点によるものであろう：

① なじみやすい機能によって，並列性が実現されていること．簡単にいえば，ふつうの手続き型の逐次プログラムが外界との間で入出力をしていたのに対し，CSP では，そのような逐次プログラムが複数個並行して走り，互いの間でも入出力をするようになっただけである．

② 言語が非常に単純であること．言語にはそれぞれ使命があって，たとえば Ada や Common Lisp が複雑だからよくないなどと，一概にいえるもの

ではない．だが，手続き型言語や Lisp（これももはや手続き型言語だ，という説が有力だが）のような成熟市場ならともかく，並行プログラミングのようにわけのわからない（とされている）ものが対象ならば，単純明快な言語の存在意義は大きい．

並行プログラミングへの第一歩がこれくらい簡単になるならば，プログラミング教育の非常に初期から，並行プログラミングを教えることができる．Pascal などで逐次プログラミングを学んで，逐次計算機上でのスピード感覚を身に付けるのも重要だが，それだけしか勉強しないと，フォン・ノイマン計算機を前提にした思考に強くしばられてしまい，それがすべてであると思い込んでしまう危険がある．並行プログラミングを初期から学ぶことは，思考のトレーニングに有効であるばかりでなく，オブジェクト指向プログラミングへのよい入門にもなるであろう．1.2 節でも触れたように，メッセージのやりとりによって交信するプロセスは，オブジェクトにほかならないのだから．

本章で紹介する言語 GHC (Guarded Horn Clauses) は，いろいろな意味で CSP の影響をうけて設計された並行プログラミング言語である．技術的な詳細は次節以下で明らかにするが，通信し合うプロセスを用いてプログラムを構成するという点や，単純であるという特徴は引き継ぎ，さらにそれを徹底しようとしている．余談だが，意味をなす長さ 3 の単語列をもって言語の名前としたのも，CSP にあやかったものである．

並行プログラミングへの導入言語として，CSP や，その実用版である Occam[28] も悪くない選択だと思うが，GHC から入るのも，次のような意味で興味深い：

- CSP と違って，各プロセスの内部にも逐次実行の概念がないという意味で徹底的に並列である．つまり GHC では，本質的に必要でない逐次性を一切使わずに，並行プログラムが書ける．このことを知るのは重要である．
- CSP の変数には，異なる値を何回でも代入できる．つまり破壊的代入ができる．一方 GHC の変数は単一代入変数であり，値を 1 回代入したら変更できない．この単一代入変数の採用で，プロセス間通信や同期のメカニズムが簡明になっている．
- 機能が原始的で，他言語にみられるいろいろな機能を統一的に説明する土台を与えてくれる．

●機能が原始的ではあるが，GHC は手続き型のアセンブリ言語と違ってプロセスを用いた抽象化の機能をはじめから備えている．

本書は，新しいプログラミング・パラダイムと銘打っているから，この点からも，非手続き型の並行言語を紹介する価値はあろう．

1.4　GHC が与える並行計算の枠組み

本節では，GHC という言語が提供する並行計算の枠組み（あるいは**並行計算モデル**）を標語的に示し，それに解説を加えてゆくことにしよう．GHC の解説記事はすでに参考文献 11)，36)，37) などがあるので，本章では特に“ものの見方”に重点をおいて話をすすめることにしたい．構文の説明やプログラム例を早く見たい人には気の毒だが，回り道と思わないでつき合ってほしい．

1.4.1　GHC における計算とは，外界との情報のやりとり（通信）である

いきなり，計算とは通信である，などという聞き捨てならぬ標語から始まった．ここでいう計算とは，GHC（でなくてもよいのだが）で書いたプログラムの実行を，外界から眺めたものを指している．多くの読者は，“計算”と聞くと，算術式やラムダ式の評価，あるいは代入文，goto 文のような文 (statement) の実行をイメージするであろう．たしかに，計算というものを内側から，つまり計算機の中に入り込んでとらえれば，このようなメカニズム的なものが見えてこよう．だが，そのような計算機の内側のことを捨象してしまうと，結局，与えられた問題やデータを処理する主体 A（プログラム）と，A に問題やデータを与えて，計算結果を受け取る主体 B（外界）との間の通信――ふつう入出力

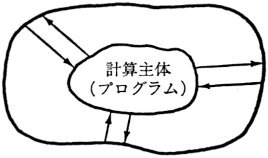


図 1.1　“計算”のイメージ

と呼ばれる——が残る（**図1.1**）．並行プログラミングやオブジェクト指向パラダイムの特徴は，このような情報の授受に注目する点である．

外界との通信は，計算の最初や最後だけでなく，計算の**途中**にも起こりうる．このことは，会話的なプログラムを考えるときに重要となる．

1.4.2 計算を行なう主体は，互いに，および外界と通信し合うプロセスの集まりである

プログラムが通信機能を持つならば，通信の相手は，別に外界でなく，並行して動いているほかのプログラムであってもかまわないはずである．とすると，複数個の，並行して動作し，互いに通信し合うプログラムを，全体として1個のプログラムとみなすこともできる．逆にいうと，1個のプログラムを，複数個の並行して動くプログラムを部品として構成することができる（**図1.2**）．

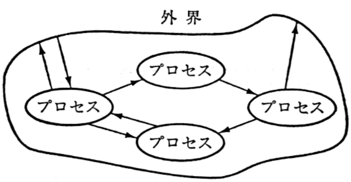


図 1.2 並行プログラムの構成

このような，ほかと並行して動き，通信機能を持つプログラムのことを，一般に**プロセス**と呼ぶ．なぜプロセスと呼ぶかというと，関数や手続きと異なり，実行の結果でなく，実行の過程（プロセス）における情報の授受が興味の対象となるからである．

1.4.3 プロセスは，停止するとは限らない

プロセスは，その実行過程，つまり外から見れば通信の履歴が主な興味の対象であり，いずれ停止するか否かということは，必ずしも重要ではない．たとえば，バンキング・システムを考えてみよう．銀行マンはともかく，現金支払機を操作する一般預金者にとっては，有限時間内にお金が出てくるかどうかが当面の関心事で，システムの停止性など，どうでもよいのである．

一方，関数型プログラミングや項書換えシステム[21]では，通常，評価が終わったときの値が問題であって，その場合，停止性は本質的である．Miranda[22]のように，遅延評価を行なう関数型言語の場合は，無限回のリダクションによって，極限において無限リストを返すような関数も有用でありうるが．

1.4.4　プロセスは，開いた系（open systems）をモデル化する

これは，プロセスが外界と通信できる，ということを言いかえたものである．開いた系というのは，閉じた系，つまり外界との相互作用なしに計算を行なう系の対立概念である．いままでに作られた多くのプログラミング言語では，外界との通信機能はad hocな形でしか与えられてこなかった．特にLispやPrologなどの（もともと）非手続き的（だったはずの）言語で，この問題は顕著であった．この意味で，開いた系の記述への適合性は，重要な特徴となりうる．

1.4.5　情報とは変数と値との結付き（結合）のことである

そろそろ，GHCに特有の話やプログラム例もまじえながら，話を具体化してゆこう．

プロセスがやりとりする情報とは，一体どのようなものであろうか？

たとえば，5という自然数を考えてみる．この5が情報であるかというと，これは単なる自然数であって，情報とはいえない．だが，「A君の一学期の国語の成績は5である」というのは立派な情報である．つまり，何か知りたいものがあって，その値（ここでは，知りたかったものは値として記号的に表現できるものとする）がわかるということが重要なのである．

GHCでは，何かわからないもの（未知数）を表わすのに変数を用いる．A君の一学期の国語の成績を表わす変数をKとしよう（本章では，変数は大文字で始める）．すると，上述の情報は，Kと5との結び付き（**結合**（binding）といい，K←5などと書く）としてモデル化することができる．変数の値に関する情報が得られることを，変数が**具体化**（instantiate）するという．

GHCの変数は，いったん値がきまると，その結び付きが変更されたり取り消されたりすることは未来永劫ない．変更されることがないという点で，GHC

の変数は手続き型言語の変数と異なる．ではPrologのような論理型言語の変数と同じかというと，Prolog の変数の値はバックトラックによって取り消されることがあるのに対し，GHC の変数の値はその心配がない．国語の成績が5だとわかったら，それを知ったプロセス（A君）は，安心してその情報を使う（たとえばプレゼントをねだる）ことができるのである．

変数の値に関する情報は，一度にまとめてくるとは限らない．たとえば,「Xは100未満の素数の列（リスト）である」という情報X←[2, 3, 5, …, 89, 97]は，より小さな情報 —— たとえば,「Xは2を先頭とするリストである」という情報と,「残りが[3, 5, …, 89, 97]である」という情報 —— に分割されてくるかもしれない．残りに関する情報も，同様に分割できる．GHC では(というか論理型言語一般の特徴だが),「Xは2を先頭とするリストである」というような不完全情報（あるいは部分情報）を，新たな変数（X1とする）を用いてX←[2|X1] と表現することができる．ここで[2|X1] は，2を先頭要素とし，残りがX1であるようなリストを表わす．X1が,あとでX1←[3|X2], X2←[5|X3] のようにきまってゆく部分である．

この例におけるXの値のように，要素の値が頭のほうから徐々にきまってゆくようなリストのことを，我々は**ストリーム**といいならわしている．ストリームについては1.5.1項そのほかでまた詳しく述べる．

1.4.6 プロセスは，結合の観測と生成を行なう

Xの階乗の値X！を計算するプロセスを考えてみよう．GHCのプロセスが外界に結果を伝える唯一の方法は，変数と値との結合を生成することである．そこで，階乗計算のプロセスは，Xと，結果を返すための変数（Yとする）を引数として，factorial(X, Y) というような形で書くことにする．引数は，外部との通信の窓口だと考えればよい．

プロセス factorial(X, Y) の仕事は，Xの値を観測し，わかったらその階乗値を計算してYに返すことである．たとえばXが5であることがわかったら，Yを120に具体化する（**図1.3**）．factorial(X, Y) を起動したとき，Xの値はすでにわかっていても，まだわかっていなくてもよい．まだわかっていなければ，factorial(X, Y) は，Xの値がわかるまで待っている．これがGHCにおける**同期**

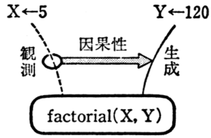


図 1.3　プロセス factorial (X, Y) の機能

(synchronization) のメカニズムである．

さて，プロセスというからには，階乗の値を1回計算して終り，ではいかにも寂しい．いろいろな自然数が次々にやってきて，それらの階乗を求めることを考えてみよう．

次々にやってくる自然数 $n_1, n_2, \cdots$ を表わすのに，X←n_1, X←n_2, … といった結合を次々に作るわけにはいかない．GHC の変数の値は，変更がきかないからである．このような場合には，ストリームを利用する．自然数のストリームを観測し，それらの階乗値のストリームを生成するようなプロセスを考えるのである．このプロセスを，factorials(Xs, Ys) としよう（**図 1.4**）．factorials(Xs, Ys) を起動しておいて，Xs←[5 |Xs1] という結合を Xs に与えると，このプロセスは Ys←[120|Ys1] という結合を生成する．次に 7! を計算したくなったら，Xs ではなくて，Xs1 に対して Xs1←[7|Xs2] という結合を与える．すると，factorials(Xs, Ys) は Ys1←[5040|Ys2] という結合を生成する．ここでもう階乗計算をやめたくなったら，Xs2←[]（[] は空リスト）という結合を与えると，factorial(Xs, Ys) は Ys2←[] という結合を生成して，その実行を終了する．

終わってみれば，結局 factorials は Xs←[5, 7] という情報に対して Ys←[120, 5040] という情報を返したことになるが，重要なことは，Xs の値（入力）が全

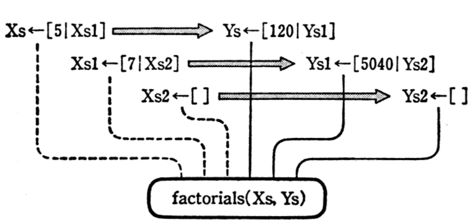


図 1.4　プロセス factorials (Xs, Ys) の機能

部きまらないうちに，Ysの値（出力）の一部をきめることができた，という点である．この性質は，ストリームのパイプライン処理（1.5.3項）に活かすことができる．

1.4.7 プロセスは，書換え規則を用いて定義する

前節の factorial や factorials の定義は，GHC ではどう書くのだろうか？

CSP をはじめとして，多くの手続き型の並行言語では，プロセスの内部の繰返し構造を，ループによって記述していたが，GHC では繰返しは再帰 (recursion) によって実現する．つまりプロセスはプロセスを用いて定義する．たとえば factorial の定義は次のようになる：

① factorial(X, Y) :- X>0 | X1 :=X−1, factorial(X1, Y1), Y :=X*Y1.

② factorial(0, Y) :- true | Y=1.

①，② のそれぞれを，**ガードつき Horn 節** (guarded Horn clause)，あるいは単に**節**という．このガードつき Horn 節の集合がプログラムとなるところが，GHC という名前の由来である．また，factorial(X, Y) のように，名前のうしろに引数が並んでいるものを，**ゴール**という．ゴールは，上の“X1 :=X−1”や“Y=1”のように演算子記法で書くこともある．前節では，factorial(X, Y) をプロセスと呼んでいたのに，なぜここではゴールと呼ぶのかと思うかもしれない．実は，ゴールといったときには factorial(X, Y) という形，つまりシンタックスに言及しており，プロセスといったときには factorial(X, Y) という形のゴールによって実現される振舞い，つまりセマンティクスに言及しているのである．

上の個々の節は，次のように読む：

① factorial(X, Y) の形のゴールがあって，Xが正整数であることがわかったら，新たに二つの補助変数（X1, Y1とする）を用意して，もとのゴールを X1 :=X−1, factorial(X1, Y1), Y :=X * Y1 の三つのゴールに置き換えてよい．

② factorial(X, Y) の形のゴールがあって，Xが0であることがわかったら，それをゴール Y=1 で置き換えてよい．

つまり，各節は，**ゴールの書換え規則**であり，縦棒の左側に，その規則が適

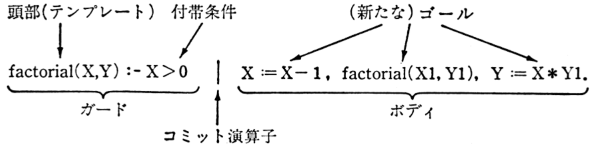


図 1.5　節（ゴールの書換え規則）の構成

用できる条件，右側に書換え後のゴールが書いてある．縦棒の左側を**ガード**，右側を**ボディ**と呼び，縦棒を**コミット演算子**と呼ぶ（**図 1.5**）．ガードの中の記号“:-”の左側を**頭部**と呼ぶ．頭部は，どういう形のゴールを書き換えるかを表わし，ガードの残りの部分（“:-”と“|”の間）は，その書換えのための付帯条件を示す．②の true は，付帯条件が空であることを示す．

先ほどプロセスとゴールの違いを強調したのは，ゴールの書換えが起きても，そのゴールによって実現されるプロセスが概念的に消えてなくなるわけではない，ということを明確にしたかったからである．GHC のプロセスが，書換え規則に従うゴールによって実現されているということは，そのプロセスを使う側には関係のないことである．プロセスを実現しているゴールは，書換え規則によって書き換えられていくので，あるプロセスを構成するゴールの数は，時々刻々変化する．

さて，節の頭部とゴールとの間では，パターン・マッチングが行なわれる．たとえば factorial(M, N) というゴールは，①の頭部 factorial(X, Y) とマッチする．そして M が正整数であることがわかったら規則①を適用して，X1 :=M−1, factorial(X1, Y1)，N :=M*Y1 に置き換えてよい．一方，factorial(M, N) は，②の頭部 factorial(0, Y) とは，Mの値が 0 であることがわかるまではマッチできない．つまり，パターン・マッチングのときは，書き換えようとするゴールのほうが，頭部よりも一般的な形をしていてはならない．

②の Y=1 というプロセスは，Yを1に具体化するもので，“=”の定義は，言語に作り付けになっている．①の X1 :=X−1 は，Xの値がわかるまで待って X−1 の値を計算し，X1 をその値に具体化する．“:=”の定義は，概念的には GHC で記述できるので，言語に作り付けになってはいないが，通常は処理系側で用意してある．“=”と“:=”の違いは，“:=”が右辺を算術式と思っ

て評価するのに対し，“＝”は評価しない点である．たとえば X1=5−1 を実行すると，X1 は 4 でなく，5−1 という形のもの（項と呼んでいる）に具体化する．“:=”の定義が，概念的には GHC で記述できるといったのは，

X :=5−1 :- true | X=4.

のような形の，算術式からその値を求める規則が，多数列挙してあると思えばよい，という意味である．

次に，factorials の定義は，下のようになる：

③ factorials([X|Xs1], Ys) :- true |
factorial(X, Y), Ys=[Y|Ys1], factorials(Xs1, Ys1).

④ factorials([], Ys) :- true | Ys=[].

③のガードは，“factorials(Xs, Ys) の形のゴールがあって，Xs が [X|Xs1] の形である，つまり長さ 1 以上のリストであることがわかったら”と読む．

①節や③節のボディのゴールの間には順序の概念がなく，並行して書換えを繰り返してよい．これが，GHC の並列性の源泉である．そうはいっても，同期の規則があるから，複数のプロセスが本当に並列に実行できるとは限らない．①では，ゴール factorial(X1, Y1) は，ゴール X1 :=X−1 が X1 の値をきめるまで書換えができないし，ゴール Y :=X*Y1 は，ゴール factorial(X1, Y1) が Y1 の値をきめるまでは掛算のしようがない．

しかし，③では事情が違う．たとえば，ゴール factorials(Xs, Ys) は，結合 Xs←[150|Xs1] を観測すると，

factorial(150, Y)
Ys=[Y|Ys1]
factorials(Xs1, Ys1)

の三つのゴールに置き換わる．これらはそれぞれ，150! の計算，出力リスト構造の作成，および残りの入力の処理を行なうプロセスであって，並列に処理できる．実際，Xs1 が [3|Xs2] だとわかったら，150! の計算と並行してゴール factorials(Xs1, Ys1) が

factorial(3, Y1)
Ys1=[Y1|Ys2]
factorials(Xs2, Ys2)

に分裂し，3! が 150! よりも先に求まってしまうかもしれない．

1.4.8　通信は，プロセス間の共有変数を用いて行なう

GHC におけるプロセス間通信は，あるプロセスが生成した結合を，別のプロセスが観測することで成立する．したがって，通信は，両プロセスの**共有変数**，つまり双方からアクセスできる変数を介して行なわれることになる．

しかしながら，この GHC の共有変数通信は，手続き型並行言語における大域変数を用いた共有変数通信とはまったく異なる．まず，1対1や1対多の通信についていうと，GHC では，変数の値を積極的に不定にしておくことができるので，「値がわかるまで待つ」という自然な形で同期を実現することができる．手続き型の並行プログラミングでは，共有変数通信とメッセージ通信が対立する概念として論じられることが多いが，GHC の共有変数通信は，同期と通信が一体となっているという点で，メッセージ通信でもある．実際，結合の生成はメッセージ送信，観測はメッセージ受信と解釈できる．

一方，手続き型並行言語では，多数のプロセスが変数を共有し，排他的にアクセスしてその内容を書き換えるということができるのに対し，GHC の変数はそういう目的には使えない．そのような共有資源を実現するには，GHC ではプロセスを用いる．プロセスによる共有資源の実現については，1.5.2項で述べる．

GHC のプロセス間通信を，メッセージ通信としてみたときの特徴は，メッセージの列がストリームというデータ構造として明示的に存在していることである．Ada のような手続き型並行言語や，オブジェクト指向言語では，メッセージの列の概念が言語に組み込まれていて，それに対する専用の操作（メッセージ送受信など）が用意されていた．GHC では，メッセージ列は，単なるリスト構造だと思って扱うことができる．これは，概念整理の観点から興味深いし，プログラムの性質を形式的に議論するときに有利であると期待できる．

1.4.9　外界も，プロセスとしてモデル化される

トイ・プログラムを別にすれば，プログラムは通常，自分で外界と通信しなければならない．表計算言語などの，特定用途の問題向き言語では必ずしもそ

うはいえないが，汎用言語では，入出力がプログラムで制御できることが必須である．GHC は，この意味で汎用言語である．

GHC では，入出力とプロセス間通信を，同一のメカニズムで実現できる．各処理系は，外界，具体的にいうと周辺機器をモデル化したプロセスを提供し，ユーザプロセスはそれとプロセス間通信をするのである．

例を挙げよう．Prologの上に筆者が作成した GHC 処理系[11]には outstream(X) という組込みのプロセスがある．これは出力端末をモデル化したもので，出力指令のストリーム X の中の各指令を順に処理する．出力指令としてはwrite(hello) や nl など，Prolog では副作用によって出力を行なうゴールであったものを用いる．たとえば，“hello” と表示して改行を行なうのに，Prolog ではwrite(hello) と nl いう “ゴール” をこの順に実行していた．GHC では，outstreamの引数 X を [write(hello), nl|X1] に具体化すればよい．

1.4.10 通信は，非同期的である

GHC のプロセス間通信では，送信側，つまり結合を生成する側のプロセスは，その結合を受信側プロセスが観測するのを待たずに，処理をすすめることができる．つまり，GHC のプロセス間通信は非同期的である．

さらに GHC では，ABCL/1[34] と違って，送った二つの情報が送った順番に受信側に届き，処理されるという保証も，計算モデルの上では存在しない．というか，もっと正確にいうと，そもそも GHC では，独立の二つの情報を送るのに，時間的順序を指定することができない．たとえば，以下に定義するプロセス p(X,Y) は，X←5 という情報と Y←6 という情報を送信するが，それを行なうプロセス X＝5 とプロセス Y＝6 がどのような順序で実行されるかはわからない：

```
p(X,Y) :- true | X=5, q(Y).
q(Y) :- true | Y=6.
```

しかしながら，通信にストリームを使えば，その中では要素は空間的に順序づけられているから，受信側で順序を取り違えることはない．つまり，複数個のメッセージを，順序を保って送りたいときは，1本のストリームを使って送ればよい：

p(Xs) :- true | Xs=[5, 6].

ABCL/1では，送信順序の保存が計算モデルの問題であるのに対し，GHCではプログラミングの問題となっているわけである．

1.4.11 プロセスのふるまいは，非決定的でありうる

切符の予約システムのように，複数台の端末から入力される要求を処理するシステムを考えてみよう．話を簡単にするために，端末は2台しかないとする．1台の端末からの要求は逐次的に発生するから，その要求列はストリームで表現するのが自然である．一方，システムは，そのようにしてできる2本のストリームの中の要求に，適当な順序を付けて，一本化して処理しなければならない．

もしこの順序づけを，両ストリームの内容だけから，一意にきめることができるならば，システムは**決定的**であるという．だが，切符の予約システムの場合は，各要求を，その内容ではなくて“時間的順序”に従って処理する必要がある．この時間的順序なるものがまた曲者で，2台の端末からほとんど同時に要求が出た場合のような，微妙なケースまで考えると，客観的に“正しい”順序などというものは，ありえないことがわかる．つまり，ほぼ同時に二つの要求がきたら，どんなに公平なシステムでも，自分の裁量でどちらかを選んで先に処理せざるを得ない．これは，このシステムの挙動が，入力だけからは本質的に定められない，つまり**非決定的**でなければならないことを示す．このようなシステムを記述することは，GHCの重要な応用の一つであるので，GHCは非決定的処理の機能を備えている．具体的には，節，つまりゴールの書換え規則の中に，適用可能なものが複数個あるとき，そのどれを適用するかを，プロセス自身の判断に委ねている．

読者の中には，上の非決定性の問題は，各要求にタイムスタンプを付ける，つまり各要求に，その送信時刻を付けて送ることで解決できると思う人がいるかもしれない．たとえば，秒単位のタイムスタンプを付ければ，秒単位の精度で，要求を送信順に処理できそうにも思える．しかしそうは問屋がおろさない．GHCでは，2台の端末（をモデル化したプロセス）とシステム（をモデル化したプロセス）の間の通信には遅れが伴ってもよいことになっている．す

ると，11時22分33秒のタイムスタンプを持つ要求rが端末Aからきても，ただちにそれを処理するわけにはいかないのである．端末Bからの要求を5秒待ってみて，11時22分32秒以前の要求がこなければ，rを処理する，というたぐいの解決法は，抽象計算モデルの世界では，一般性も説得力も持たない．

1.5 もう少し具体的なパラダイム

前節では，GHCの提供する並行計算モデルの基本について解説した．本節では，そのような計算モデルを用いて，具体的にどのようなプログラミング上の概念が実現できるかを示す．

1.5.1 ストリームと双方向通信

前節でも触れたが，ストリームとは頭のほうから徐々に生成され，消費されるリスト構造のことである．GHCでは，ストリーム型のようなものが，言語に作り付けになっているわけではない．GHCのストリームは，静的には単なるリストであり，上述のような作り方，使い方をするリストを特にストリームと呼びならわしているにすぎない．

ストリームの概念は，Miranda[22]，KRC[12]をはじめとする関数型言語にもみられる．関数型言語の場合，ストリームを頭のほうから生成するためには，ふつう遅延評価というメカニズムを用いる．一方GHCでは，単一代入変数を利用して，リストの残りをとりあえず未定にしておいて先頭の要素を先にきめることでストリームを実現している．

ストリームに関して重要なことは，生成プロセスが止まらない場合は，極限において無限長のリストができるということである．このことの影響が，一つにはプログラムの理論的扱いに現われる．一般に，プログラム（の断片）の性質についての議論は，停止性と，止まったときの結果の2段構えで行なうことが多かった．だが並行プログラムの場合は，止まらないプログラムから得られる結果も議論できなければならない（1.4.3項参照)．そのような議論においては，ストリームの長さに関する帰納法が一般には使えないことになる（ためしに，二つのプログラムの生成する無限ストリームが同じであるかどうかを論ず

るにはどうしたらよいかを考えてみてほしい)．これに対する一つの代案は，ストリームの**観測精度**に関する帰納法を使う方法である．つまり，二つの無限ストリームが同じであるということを，「頭から要素を比較していったとき，有限回の比較では違いが発見されないこと」と定義し，比較の回数に関する帰納法を用いるのである．GHC の場合は，Miranda のような関数型言語と違って，ストリームが双方向通信にも使えること（下記参照），非決定性を記述できることなどのために，実はもう少し話が複雑になるが，プログラミングの性質を議論するために観測精度の概念を用いることができるという点は共通である．この論法で，止まらないプログラムに関して現在考えられているさまざまな性質をすべて扱えるわけではないが，多くの“素直”な性質は，この方法で扱うことができる．

1.4.10 項に述べたように，ストリームによるプロセス間通信は，完全な非同期通信である．ストリームの生産者（送信プロセス）は，消費者（受信プロセス）に無関係に要素の値をきめてゆくことができ，消費者もまた，生産者を追い抜かないという制限の下で，マイペースでその値を使っていってよい．つまり，ストリームは非有界バッファ（unbounded buffer）の役割を果たしている．

GHC のストリームの重要な特徴は，生産者から消費者への単方向通信に使えるばかりでなく，**返信を伴うような双方向通信も1本のストリームでできる**点である．つまり，メッセージの受信者から“返事”がほしい場合，送信メッセージに変数を含めておき，受信者にその変数を具体化してもらうのである．

例を挙げよう．端末のキーボードからデータを入力したい場合，前述の Prolog 上の GHC 処理系では，instream(X) という組込みプロセスを利用する．X を [read(T)|X1] に具体化すると，instream(X) は端末からデータを1個読み込み，その値をTと結合する．read(T) のように，変数を含むメッセージを，**未完成メッセージ**（incomplete message）という．未完成メッセージに対して返答を出し，メッセージを“完成”させるのは，受信側の役割である．

instream(X) には，実は出力メッセージを送ることもできる．たとえば，X を [write('more?'), nl, read(T)|X1] に具体化すれば，“more?” というプロンプトが出て，改行したあとで読込みが起きる．instream に write('more?') のような出力指令を送れるようにしたのは，出力指令と入力指令を，同じストリー

ムの中に並べて順序づけることにより，ディスプレイへのプロンプト出力と，データの読込みとの時間的前後関係を保証できるようにするためである．1.4.9項で説明した outstream が，出力端末をモデル化したものであったのに対し，instream は，入出力端末をモデル化したものであるといえる．

未完成メッセージは，プロセス間の双方向通信の記述を大変容易にする．双方向通信が容易に書けるということは，オブジェクト指向のプログラミングや会話的プログラムの記述に大変重要である．双方向通信は，もちろん単方向通信のストリームを2本使ってもできるが，プログラムが繁雑になるのは否めない．

1.5.2 履歴のあるオブジェクトの表現

未完成メッセージを用いると，履歴を持つオブジェクトを容易に記述できる．オブジェクトをプロセスとして表現し，そのプロセスにつながるストリームに指令の列を与えるのである．たとえば，スタックをプロセスとして定義すると，次のようになる：

```
stack(Xs) :- true | stk(Xs, []).
stk([push(T)|Xs1], Ls) :- true | stk(Xs1, [T|Ls]).
stk([pop(T)  |Xs1], [L|Ls1]) :- true | T=L, stk(Xs1, Ls1).
stk([pop(T)  |Xs1], []) :- true | T=error, stk(Xs1, []).
stk([], Ls) :- true | true.
```

引数 Xs は，スタックに対する指令列を表わすストリームである．実際にスタックを使う場合は，未定義変数（S とする）を用意して，stack(S) というプロセスを起動する．そして，S←[push(5), push(4), pop(X), pop(Y), pop(Z)] という結合を与えると，X=4, Y=5, Z=error という答が返ってきて，スタック・プロセスは終了する．stack を異なる引数で2回起動すれば，二つの独立なスタック・プロセスができる．

一般に，関数型言語や論理型言語など，変数への破壊的代入を許さない言語で，履歴（状態）を持つオブジェクトを表現することは困難だと考えられているが，GHC では，書換え規則で定義したプロセスと双方向通信を用いることによって，その問題を解決している．

上のスタックを，複数のプロセスが排他的にアクセスする共有資源とするときは，それらのプロセスからの指令列を併合（merge）して，一本化しなければならない．ストリームの併合は，基本的には，次のように定義される併合プロセスを用いて行なう：

```
merge([A|Xs1], Ys, Zs) :- true | Zs=[A|Zs1], merge(Xs1, Ys, Zs1).
merge(Xs, [A|Ys1], Zs) :- true | Zs=[A|Zs1], merge(Xs, Ys1, Zs1).
merge([], Ys, Zs) :- true | Zs=Ys.
merge(Xs, [], Zs) :- true | Zs=Xs.
```

併合プロセスを用いて，指令列を一本化するのは，GHCにおける多対1の通信の基本テクニックである．ABCL/1をはじめとして，多対1のメッセージ通信を許す多くの言語では，指令列の一本化機能は言語に組み込まれているが，GHCは原始的であるため，そのような一本化操作が介在している以上，そのことを明示的に書かなければならない．これも一つの見識だと思うのだが，いかがであろうか．

そうはいっても，多対1の通信を多用するアプリケーションでは，併合プロセスをいちいち書くのは負担であろう．また，その処理効率も重要である．そこで，ストリームの併合については，プログラミングの効率と実行効率の両面から，研究がすすめられている．

1.5.3　データ駆動計算と要求駆動計算

データ駆動計算（data-driven computation）とは，関数の値やプロセスの出力などを計算しようとするときに，必要な入力データが揃っていれば，その出力をほかの関数やプロセスが必要としているかどうかにかかわらずに計算をすすめる計算法をいう．

一方，要求駆動計算（demand-driven computation）とは，関数の値やプロセスの出力などを積極的に計算せず，それらを利用する関数やプロセスから要求がきてはじめて計算する計算法をいう．

関数型言語では，この2種（もっと細かく分類することもあるが）の計算法の違いは，引数の評価のメカニズム——call by value, call by need など——の違いとして，計算モデル，つまり言語の操作的意味論の中で論じられてきた．こ

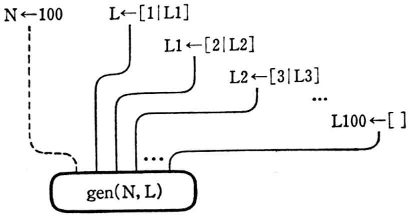


図 1.6 自然数列の生成（データ駆動版）

れに対して GHC では，データ駆動と要求駆動の違いは，同じ（GHC の）計算モデルの下での，プログラミング技法の違いとして実現している．

例を挙げよう．自然数の列 [1, 2, …, N] を積極的に生成するプロセスの定義は，次のように書ける：

```
gen(N, L) :- true | g(0, N, L).
g(I, N, L) :- I<N | I1 :=I+1, L=[I1|L1], g(I1, N, L1).
g(I, N, L) :- I≧N | L=[].
```

プロセス gen(N, L) は，N の値さえわかれば，ストリーム L を自律的に生成してゆく（図 1.6）．これに対して，次の定義を考えてみよう：

```
genlazy(L) :- true | gl(0, L).
gl(I, [J|L1]) :- true | I1 :=I+1, J=I1, gl(I1, L1).
```

プロセス genlazy(L) は，L の値を自律的に生成できない．genlazy(L) は，ストリーム L に関して消費者になっているのである．ではどうやって genlazy(L) を駆動するかというと，L を外部から相異なる変数のストリーム [K1, K2, K3, … に具体化してやる．すると，genlazy は K1←1, K2←2, K3←3, … という結合を生成するのである（図 1.7）．

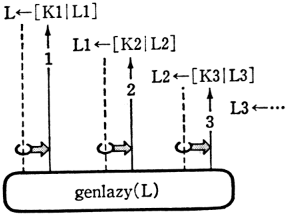


図 1.7 自然数列の生成（要求駆動版）

```
primes(Max,Ps) :- true | gen(1,Max,Ns), sift(Ns,Ps).

gen(N,Max,Ns) :- N< Max | N1:=N+1, Ns=[N1|Ns1], gen(N1,Max,Ns1).
gen(N,Max,Ns) :- N>=Max | Ns=[].

sift([P|Xs],Zs) :- true | Zs=[P|Zs1], filter(P,Xs,Ys), sift(Ys,Zs1).
sift([],    Zs) :- true | Zs=[].

filter(P,[X|Xs],Ys) :- X mod P=:=0 |              filter(P,Xs,Ys).
filter(P,[X|Xs],Ys) :- X mod P=\=0 | Ys=[X|Ys1], filter(P,Xs,Ys1).
filter(P,[],    Ys) :- true        | Ys=[].
```

図 1.8　素数列生成プログラム（データ駆動版）

お気づきかもしれないが，gen ではストリームを単方向通信に使っていたのに対し，genlazy では双方向通信に使っている．genlazy への送信データ Ki(i=1, 2, …) は，返信用の変数だけからなる未完成メッセージであって，genlazy に対して，次の値を計算するように要求していると考えることができる．GHC の要求駆動計算では，計算結果を要求するプロセスがストリームの骨格（リスト構造）を作成し，計算をするプロセスがその中身を埋めてゆく．1本のストリームが，二つのプロセスの共同作業で作られるわけである．

さて，上の自然数列生成プロセスを用いて，素数列を生成するプログラムを作ってみよう．まずデータ駆動版は，**図 1.8** のようになる．プロセス primes(Max, Ps) は，ただちに二つのプロセス，つまり自然数列生成プロセス gen(1, Max, Ns) と，素数以外をふるい落とすプロセス sift(Ns, Ps) を生成する．後者は，素数Pから始まり，P未満の素数の倍数がすでにふるい落とされている数列 Ns から，素数列 Ps を作り出すものであり，素数が1個見つかるごとに，その倍数をふるい落とす filter プロセスを生成している．filter プロセスはストリームによって一列につながっていて，パイプライン処理によって，gen の生成する自然数列から，各素数の倍数を並行してふるい落としてゆく，filter の定義の中で，“X mod P=:=0” は，“X が P で割り切れることがわかる”，“X mod P=\=0” は，“XがPで割り切れないことがわかる”，とそれぞれ読む．

次に，要求駆動版は**図 1.9** のようになる．プログラムの基本的構造はデータ駆動版と同じだが，gen, sift, filter の各定義で，ストリームの骨格の流れる向きが逆になっている．たとえば，filter(P, Xs, Ys) というプロセスがあったとすると，Ys から，「Xs の要素でPの倍数でないものが欲しい」という要求がくる．

```
primes(Ps) :- true | gen(1,Ns), sift(Ns,Ps).

gen(M,[N|Ns1]) :- true | M1:=M+1, N=M1, gen(M1,Ns1).
gen(M,[]     ) :- true | true.

sift(Xs,[Z|Zs1]) :- true |
        Xs=[Z|Xs1], filter(Z,Xs1,Ys), sift(Ys,Zs1).
sift(Xs,[]     ) :- true | Xs=[].

filter(P,Xs,[Y|Ys1]) :- true | Xs=[X|Xs1], filter2(P,X,Xs1,Y,Ys1).
filter(P,Xs,[]     ) :- true | Xs=[].
filter2(P,X,Xs1,Y,Ys1) :- X mod P=:=0 |      filter(P,Xs1,[Y|Ys1]).
filter2(P,X,Xs1,Y,Ys1) :- X mod P=\=0 | Y=X, filter(P,Xs1,Ys1    ).
```

図 1.9 素数列生成プログラム（要求駆動版）

```
test :- true | instream(IOs), ask(IOs,Ps), primes(Ps).

ask(IOs,Ps) :- true | IOs=[read(M)|IOs1], ask2(IOs1,Ps,M).
ask2(IOs1,Ps,M) :- M > 0 | Ps=[P|Ps1], ask3(IOs1,Ps1,M,P).
ask2(IOs1,Ps,M) :- M=:=0 | ask(IOs1,Ps).
ask2(IOs1,Ps,M) :- M < 0 | IOs1=[], Ps=[].
ask3(IOs1,Ps1,M,P) :- true |
         IOs1=[write(P),nl|IOs2], M1:=M-1, ask2(IOs2,Ps1,M1).
```

図 1.10 素数列生成プログラム（要求駆動版）の駆動プログラム

すると，filter(P, Xs, Ys) は，ストリームXsの“次”の値を，Pの倍数でないものが見つかるまで，前段のプロセスに要求し続ける．

図 1.10 は，この要求駆動版の素数列生成プロセス primes(Ps) を駆動するプログラムの一例である．トップレベルのプロセス test を実行すると，すぐに三つのプロセスができるが，主導権を持っているのはプロセス ask(IOs, Ps) である．ask(IOs, Ps) は，端末から整数Mを読み込み，M≧0 ならば次のM個の素数を要求，表示し，また次の個数を端末に聞きにいく．M<0 ならば，IOs と Ps を閉じることにより instream(IOs) と primes(Ps) を終了させ，自らも終了する．図 1.11 に実行例を示す．

この素数列生成プログラムの二つの版を，Miranda で書いたもの[22] と比べてみると，GHC で書いたもののほうがかなり長い．これは，GHC の提供している機能のほうが，より原始的だからである．たとえば GHC 版では，要求駆動計算の要求も，プログラム上で明示的に扱っている．Miranda の素数列生成プログラムは疑いもなく高級言語のプログラムであるが，GHC の素数列生成プログラムは，それに比べればかなりアセンブラ的であるといえる．

```
| ?- ghc test.
|: 5.
2
3
5
7
11
|: 0.
|: 7.
13
17
19
23
29
31
37
|: -1.
500 msec.
```

図 1.11　素数列生成プログラム（要求駆動版）の実行

なお，ここに述べた要求駆動計算と類似の技法を用いると，有界バッファ (bounded buffer) 通信も GHC で記述できる[11] ことを付記しておこう．

1.5.4 モジュラリティと差分プログラミング

プロセスとストリームを用いたプログラミングを，programming in the large の観点から少し眺めてみよう．

前項の素数列生成プログラム（データ駆動版）を修正して，双子素数（隣り合う奇数の対で，そのどちらもが素数であるもの）の列を生成するプログラムを作ることになったとする．このとき，primes の定義にはまったく手を入れなくてよい．素数列を入力として，双子素数列を出力とするプロセスを**外づけ**すればよいのである (**図 1.12**)．

このように，少し異なる結果がほしい場合は，その差に相当する変換作業をプロセスとして定義して外づけすればよい．これが，プロセス指向の並行プログラミングにおける**差分プログラミング** (differential programming) の方法である．差分プログラミングができるということは，プログラムのモジュラリティ，

```
twins(Max,Ts) :- true | primes(Max,Ps), twin(Ps,Ts).

twin([P1,P2|Ps2],Ts) :- P2=\=P1+2 | twin([P2|Ps2],Ts).
twin([P1,P2|Ps2],Ts) :- P2=:=P1+2 | Ts=[(P1,P2)|Ts1],twin([P2|Ps2],Ts1).
twin([_],        Ts) :- true      | Ts=[].
twin([],         Ts) :- true      | Ts=[].
```

図 1.12　双子素数列生成プログラム

汎用性，再利用性にとって重要である.

GHC でプログラムを書くときは，その内容をなるべく多くの，意味のある部品（プロセス）に分けて定義し，ストリームを用いたネットワーキングによって全体を組み上げると，プログラムがわかりやすくなる．小さなプロセスを作りすぎるとプロセス間通信が増え，効率が低下すると心配する人もあろうが，それは処理系作成技法やプログラム変換技法で解決すべき性質の問題であり，プログラミング・スタイルを犠牲にして解決すべきではない．複数個の小さなプロセスをプログラム変換で融合する技法は，参考文献 39) を参照してほしい.

1.5.5 プロセスによるデータ表現

プロセスはほかと並行して動作し，ほかと通信し，また履歴（つまり状態）を保持することができる．すると，比較的小容量のデータを持つ多数個のプロセスをストリームによって構造化すれば，並列アクセスの可能性を持った，大きなデータ構造ないしはデータベースを構築できる可能性がある.

たとえば，CSP の論文[19]には，整数の集合をプロセスの配列によって表現する例が出ている．このプログラムを GHC に翻案してみよう（**図 1.13**).

このプログラムでは，集合は，**図 1.14** のように，プロセス element とプロセス emptyset からなる一次元的なプロセス構造で表現している．プロセス element (Is, Os, N) は，ちょうど 1 個の整数 N を管理しており，Is は前段の，Os は後段のプロセスにつながっている．これらのプロセスは，N の値の小さい順に整列してあり，そのうしろに末端のプロセス emptyset がつながる (図 1.14)．この集合プロセスを利用するときは最初に未定義変数 (S とする) を用いて emptyset(S)

```
emptyset([has(N,R)|Is])    :- true | R=false,         emptyset(Is).
emptyset([insert(N)|Is])   :- true | element(Is,Os,N), emptyset(Os).
emptyset([])               :- true | true.

element([has(M,R)|Is],Os,N)   :- M < N | R=false,              element(Is,Os, N).
element([has(M,R)|Is],Os,N)   :- M=:=N | R=true,               element(Is,Os, N).
element([has(M,R)|Is],Os,N)   :- M > N | Os=[has(M,R)|Os1],    element(Is,Os1,N).
element([insert(M)|Is],Os,N)  :- M < N | Os=[insert(N)|Os1],   element(Is,Os1,M).
element([insert(M)|Is],Os,N)  :- M=:=N |                       element(Is,Os, N).
element([insert(M)|Is],Os,N)  :- M > N | Os=[insert(M)|Os1],   element(Is,Os1,N).
element([],             Os,N) :- true  | Os=[].
```

図 1.13　プロセスによる集合の実現

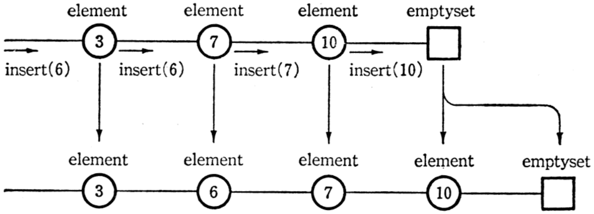


図 1.14　集合への要素の追加

を起動し，S に insert メッセージや has メッセージの列を送ればよい．

各プロセスは，メッセージ insert(N) と has(N, R) に対して適切な行動をとる．集合の要素数が増える場合の処理，つまり新たな element プロセスの作成は，emptyset の第2節で行なっている．element の第4節は，“自分が持っている整数Nより小さな整数Mの挿入要求がきたとき，Nを手放して次のプロセスに送り，自分は新たにMを持つ”という規則を表わしている．

この例題は，たかが集合を管理するのにプロセスとストリーム通信を多用しており，とても実用的には見えないかもしれない．だが，そういって一笑に付してしまうのは早い．この技法には，次のような重要な示唆が含まれている：

① プロセス網は，並行計算だけでなくデータ構造を表現するのにも使用できる．

② プロセス網としてデータ構造を表現すると，その検索，修正に並列性を活かすことができる．上の集合の例では，各メッセージがパイプライン的に処理できる．

この技法を現実のものにするには，プロセスとストリームを非常に効率的に実現しなければならないが，それに成功すれば，この技法もパラダイムとして確立してゆくと期待される．参考文献 40) は，それを目指した研究の第一歩である．

1.6　歴史的背景と文献案内

前節までに紹介した考え方や技法は，もちろん完全に GHC 独自のものとい

うわけではない．多くは，GHCに至るまでの並行プログラミングの研究成果を反映したものである．本節では，それらの成果のうち，GHCに深く影響を与えたものを，文献案内もかねて紹介しつつ，歴史を振り返ってみよう．

冒頭で，GHCは，論理プログラミングにおける並列性の研究から生まれた言語であると述べた．論理プログラミングは，Kowalskiの論文[26]によって，パラダイムとして提示され，その後目覚ましく発展した分野である．この論文はすでに，論理プログラムの並列実行の可能性を示唆しており，実際それが，GHCにおける徹底した並列実行の可能性の源になっている．またGHCは，単一化（unification）や導出原理（resolution principle）[30]など，基本計算メカニズムの多くを論理プログラミングからうけついでいる．

一方，互いに通信しあうプロセスを部品として並行プログラムを作るという考え方は，KahnとMacQueenの論文[25]およびHoareの論文[19]で，言語とともに示された．KahnとMacQueenの言語は，POP-2をベースに作られたものである．非同期通信とプロセスの動的生成を許しているが，プロセスのふるまいは（意図的に）決定的である．一方Hoareの言語CSPは，Dijkstraのguarded commands[7]をベースにしている．こちらはプロセスの動的生成，再帰呼出し，非同期通信のような，動的メモリ管理を要する機能を一切排しているが，プロセスのふるまいは非決定的でありうる．上に挙げた文献は，いずれもきわめて有名なもので，10年以上たった今でも味わい深く，教えられることが多い．

論理プログラミングの上でプロセス指向のプログラミングを考察した研究としては，van Emdenとde Lucena Filhoのもの[8]や，ClarkとGregoryのRelational Language[4]が初期のものである．前者は，KahnとMacQueenの計算モデルが論理プログラミングの上で実現できることを示し，論理プログラムにプロセス的解釈を与えた．一方Relational Languageは，KahnとMacQueenが示した非同期通信や動的プロセス網の機能と，CSPのような非決定性とを合わせ持つ言語を具体的に示したもので，その後に設計されたConcurrent Prolog, PARLOG, GHCの基本的枠組みを与えたという意味で重要である．

ShapiroのConcurrent Prolog[32),33]は，read-only annotationという同期メカニズムを用いて，未完成メッセージによる双方向通信の手法を提示し，数多くのプログラム例でその有効性を示した．履歴依存オブジェクトが自然に表現でき

ることを示した意義は大きい．一方 Relational Language を設計した Clark と Gregory は，その後継言語 PARLOG[5),14)] で，未完成メッセージの技法が，入出力モードという，より簡単な同期メカニズムで実現できることを示した．GHC[11),37),38)] は，ICOT の並列推論マシンの核言語を，Concurrent Prolog をたたき台として検討する過程で生まれたもので，同期メカニズムをはじめとする諸概念を整理して言語を単純化したことを最大の特徴とする．枚田の Oc[18)] は，GHC をさらに単純化したもので，本章に示した各技法は，実は Oc にも適用できる．Foster と Taylor は，これらの諸言語の研究成果を手際よく採り入れた商用ベースの実用言語 Strand（STReam AND parallelism の略）を設計した[9)]．またSaraswat らは，これらの言語における（項を用いた）情報の表現を一般化，抽象化し，cc (concurrent constraint) と呼ばれる言語族を提案している[31)]．

このほかにも，触れておくべき関連研究は多いが，それについては上に掲げた文献や，論理プログラミングや並行プログラミングに関係する会議論文集，論文誌を手がかりにしてあたってほしい．本節で紹介した流れに属する研究成果は，International Conference on Logic Programming (ICLP) や North American Conference on Logic Programming (NACLP，旧称 Symposium on Logic Programming) で発表されることが多い（これらの最近の論文集は，MIT Press から出ている）．並行ないしは並列プログラミングをもっぱら扱う論文誌としては，International Journal on Parallel Programming (IJPP, Plenum Press 刊) などがあり，会議としては ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming (PPoPP) などがある．

だが，並行プログラミングについての重要な論文や解説は，並行プログラミングに分野を限定していないさまざまな論文誌や会議で発表されることが非常に多い．このような論文誌や会議には，上の ICLP や NACLP のような，特定のプログラミング・パラダイムに関するものと，IEEE Computer や ACM Transactions on Programming Languages and Systems (TOPLAS)，ACM Symposium on Principles of Programming Languages (POPL) のような，プログラミング（言語）や計算機科学一般に関するものとがある．

並行 Lisp，並行オブジェクト指向言語など，論理プログラミング以外のパラダイムから発生した並行言語については，1.1 節に挙げた文献1)，13)，15)，16)，

24),34),41) を参照してほしい. Springer-Verlag の Lecture Notes in Computer Science シリーズからは, 並行プログラミングの理論的側面に関するワークショップや会議の論文集が数多く出版されている.

並行プログラミングを扱った教科書としては, 網羅的な 10)や, 理論的な 2) などがある. また, プログラミング以前の, 並行計算の基礎に関しては, Hoare[20] や Milner[27] による優れた教科書がある. 文献 20) は文献 19) と同じ表題 (Communicating Sequential Processes) を持つが, 内容は非常に異なっていて, 並行プロセスの数学的モデルを提案し, 論じている. このモデルは, Theoretical CSP (TCSP) と呼ばれることが多い. 文献 27) は, もう一つの著名な数学的モデルである CCS (Calculus of Communicating Systems) をわかりやすく解説している.

1.7 並行プログラミングと効率

せっかく並行プログラムを書くからには, 書いたプログラムが並列計算機の上で効率良く走ってほしいものである.

GHC の計算は, ゴールの書換えによって進む. これを並列に実行する最もふつうの方法は, 異なるプロセスに属するゴールの書換えを, 別々のプロセッサで同時に進める方法であろう (本節は, 汎用マルチプロセッサ・システムによる並列処理を想像しながら書いている). しかし, GHC のプロセスは互いに通信しあうから, その手間も考えなければならない. 二つのプロセスを, 一つのプロセッサの中で擬似並列処理すれば, 物理的並列性がない代わりに, プロセス間通信にかかる手間も小さい. 一方, それらを, 二つのプロセッサで並列実行させようとすると, 両者が独立ならば計算時間が短縮できるが, プロセス間通信は擬似並列処理の場合よりもはるかに高くつく. さらに, その二つのプロセスの仕事が十分大きくないと, 複数のプロセッサに仕事を分散させる手間 (これも通信の手間のうちである) に見合わない.

プロセッサ間通信の能力をはかる尺度には, latency (おくれ) と throughput (単位時間当りの転送量) とがあるが, 共有メモリによらないプロセッサ間通信の場合, 特に latency の問題が深刻である. つまり, 小さなデータを送るこ

とは，一般に非常に不利である．

プログラムがどれくらい並列実行に適するかは，以下のように定性的に議論できる．まずプロセス間の関係を，次の三つに分類する：

① 無通信，または疎な通信：たとえば，クイックソートで，“大きなデータ”の列と“小さなデータ”の列をそれぞれソートするプロセス間の関係．

② 密な単方向通信：たとえば，データ駆動型の素数列生成プログラム（1.5.3項）における，filter プロセスどうしの関係．

③ 密な双方向通信：たとえば，要求駆動型の素数列生成プログラムにおける，filter プロセスどうしの関係．

①→②→③の順に，プロセス間の関係は密になり，並列実行は困難になる．②では，通信データをまとめて送れるので，プロセッサ間通信を，throughput を重点に設計すれば，並列実行は有効である．しかし③では，通信データが小刻みになり，また複数のプロセスが互いに相手の情報を待ち合わせることになるので，プロセス単位の並列処理はむしろ有害ですらある．

結局，プログラムが並列処理に向くかどうかは，全体の計算を，局所性が高く，しかも十分に仕事量のあるプロセスに分割できるかにかかっている．並行プログラム，並列アルゴリズムを設計するときは，このことに十分注意しなければならない．このとき，並列度の大小は，それだけではプログラムのよさを評価する基準にはなりえないことに注意する必要がある．なぜなら，一般に，計算複雑度を小さくする工夫をすればするほど，並列実行には向かなくなる傾向があるからである．

効率の観点からもう一つ重要なことは，**見込み計算**（speculative computation）の考え方をうまく活用することである[16),29)]．一般に，無駄な計算をせずに処理系の並列性を活かすことができれば，それにこしたことはない．だが，並列性を活かすには，「必要最小限の計算しかしない」という態度を改めなければならないことがよくある．要求駆動型の素数列生成プログラムもこれに該当する（ではどういう解決法がありうるか，考えてみてほしい）．AI における探索問題でも，同じことがいえる．たとえば，アルファ・ベータ探索を行なうのに，探索木の枝どうしが頻繁に評価関数の交信をすれば，無駄な探索は少なくなる

が，通信量が増え，また並列性が損なわれることになる．逆に，評価関数の交信をたまにしか行なわないと，無駄な計算をしていることに長い間気づかない，ということが起きうる．見込み計算とは，役に立つことを期待して行なうが，無駄になる可能性もある計算のことである．この概念は，従来の逐次計算機においても，低いレベルの処理ではしばしば利用されてきた（例：命令の先読み）が，通常のプログラミングで意識されることはそれほど多くなかった．だが，並列処理では，これをうまく導入し，かつ制御することが効率上大変重要となるであろう．

GHC プログラムは，並列計算機の上でどう実行すべきかを何も指定していないという意味で，**並行**プログラムではあっても，（第 1.1 節に示した意味での）**並列**プログラムとはいえない．だが，GHC プログラムに，プロセスをどのように並列実行すべきかの指定を付け加えれば，並列プログラムと呼べるものが得られよう．GHC をベースに実用言語として設計された KL1[3] では，そのような指定ができるようになっている．プログラミング言語では，制御構造が重要な役割を持っているが，並列言語としての KL1 の大きな特徴は，制御のうちプログラムの正当性，つまり“並行”にかかわる部分と，効率，つまり“並列”にかかわる部分を明確に分離したことである．

1.8 まとめ

並行プログラミングの基本について，GHC を用いて解説してきた．本解説で示した並行プログラミングの考え方は，“GHC 的なものの見方”を強く反映したものだが，もとよりこれが並行プログラミングのすべてであるというわけではない．興味のある方は，ほかの言語を用いた並行プログラミングとの比較検討を試みてほしい．

並行プログラミングに親しむためには，プログラムを実際に走らせてみるのが一番よい．最も手軽なのは，前述の Prolog 上の GHC 処理系で，参考文献 11）に，ソースプログラムを含めた詳細が公開されている．より本格的な処理系としては，C で書いた PDSS（PIMOS Development Support System）と呼ばれる処理系[17] があって，UNIX 4.2（および 4.3）BSD 上で動く．また並列処理系

としては，ICOT で開発されたマルチ PSI[35] の上で KL1 が動いており，その上で並列プログラムの研究開発が活発に行なわれている．

比較的小さなプロセスを構成要素としてプログラムを作るという考え方は，手続き型プログラミングの世界でも，lightweight process あるいはスレッド (thread) という名前とともに普及しつつある．GHC はこの lightweight process の考え方を徹底させたものといってよい．ただし，徹底させた分だけ，プロセスのインプリメンテーションも徹底的に効率化する必要がある．

並行・並列プログラミングを発展させるためには，プログラミング・パラダイムや言語の開発以外にも，しなければならないことがたくさんある．使える並列処理系の開発，ちゃんとしたプログラミング環境やデバッグ環境の整備，役に立つ理論の構築など，重要なものがめじろ押しである．また，プログラミング・パラダイムも，“このような問題にはこのような技法を使う”というレベルまで具体化してゆく必要がある．多くの人に，この新しく大きな分野に取り組んでいただきたいと思う．

参考文献

1) Carriero, N. and Gelernter, D.: Linda in Context, *Comm. ACM*, Vol. 32, No. 4, pp. 444–458, 1989.
2) Chandy, K. M. and Misra, J.: *Parallel Program Design*, Addison–Wesley, 1988.
3) Chikayama, T., Sato, H. and Miyazaki, T.: Overview of the Parallel Inference Machine Operating System (PIMOS), in *Proc. Int. Conf. on Fifth Generation Computer Systems '88*, ICOT, Tokyo, pp. 230–251, 1988.
4) Clark, K. L. and Gregory, S.: A Relational Language for Parallel Programming, in *Proc. ACM Conf. on Functional Programming Languages and Computer Architecture*, ACM, pp. 171–178, 1981. Also in 33), Vol. 1, pp. 9–26.
5) Clark, K. L. and Gregory, S.: PARLOG: Parallel Programming in Logic, *ACM Trans. Prog. Lang. Syst.*, Vol. 8, No. 1, pp. 1–49, 1986.
6) Dijkstra, E. W.: Co-operating Sequential Processes, in *Programming Languages*, Genuys, F. (ed.), Academic Press, pp. 43–112, 1968.
7) Dijkstra, E. W.: Guarded Commands, Nondeterminacy and Formal Derivation of Programs, *Comm. ACM*, Vol. 18, No. 8, pp. 453–457, 1975.
8) Emden, M. H. van and Lucena Filho, G. J. de: Predicate Logic as a Language for Parallel Programming, in *Logic Programming*, Clark, K. L. and Tärnlund, S.-Å. (eds.), Academic Press, pp. 189–198, 1982.
9) Foster, I. and Taylor, S.: Strand: A Practical Parallel Programming Language, in *Proc. 1989 North American Conf. on Logic Programming*, The MIT Press, pp. 497–512, 1989.

10) Filman, R. E. and Friedman, D. P. (雨宮，尾内，高橋訳)：協調型計算システム：分散型ソフトウェアの技法と道具立て，マグロウヒルブック，1986.

11) 淵 一博監修，古川康一，溝口文雄共編：『並列論理型言語 GHC とその応用』，知識情報処理シリーズ第 6 巻，共立出版，1987.

12) 淵 一博，黒川利明編著：『新世代プログラミング』，共立出版，1986.

13) Goldman, R. and Gabriel, R. P.: Qlisp: Experience and New Directions, in *Proc. ACM/SIGPLAN PPEARLS*, Sigplan Notices, Vol. 23, No. 9, pp. 111–123, 1988.

14) Gregory, S.: *Parallel Logic Programming in PARLOG*, Addison-Wesley, 1987.

15) Halstead, R. H. Jr.: Multilisp: A Language for Concurrent Symbolic Computation, *ACM Trans. Prog. Lang. Syst.*, Vol. 7, No. 4, pp. 501–538, 1985.

16) Halstead, R. H. Jr.: Parallel Symbolic Computing, *IEEE Computer*, Vol. 19, No. 8 pp. 35–43, 1986.

17) 平野喜芳，中越靖之，西崎慎一郎，宮崎芳枝，宮崎敏彦，近山 隆：汎用計算機上の KL1 処理系—PDSS—，in *Proc. Logic Programming Conference '89*, ICOT, Tokyo, pp. 193–202, 1989.

18) 枚田正宏：並列記号処理言語 Oc とその自己記述，コンピュータソフトウェア，Vol. 4, No. 3, pp. 41–64, 1987.

19) Hoare, C. A. R.: Communicating Sequential Processes, *Comm. ACM*, Vol. 21, No. 8, pp. 666–677, 1978.

20) Hoare, C. A. R.: *Communicating Sequential Processes*, Prentice-Hall International, 1985.

21) 井田哲雄編：『新しいプログラミング・パラダイム』，第 8 章 項書換えシステムと完備化手続き（大須賀昭彦），pp. 165–186，共立出版，1989.

22) 井田哲雄編：『新しいプログラミング・パラダイム』，第 7 章 Miranda（加藤和彦），pp. 139–163，共立出版，1989.

23) 井田哲雄編：『新しいプログラミング・パラダイム』，第 5 章 オブジェクト指向（柴山悦哉），pp. 91–115，共立出版，1989.

24) Ito, T. and Halstead, R. (eds.): *Parallel Lisp: Languages and Systems——Proc. U.S./Japan Workshop on Parallel Lisp*, Lecture Notes in Computer Science 441, Springer-Verlag, 1990.

25) Kahn, G. and MacQueen, D. B.: Coroutines and Networks of Parallel Processes, in *Proc. IFIP '77*, North-Holland, pp. 993–998, 1977.

26) Kowalski, R.: Predicate Logic as Programming Language, in *Proc. IFIP '74*, North-Holland, pp. 569–574, 1974.

27) Milner, R.: *Communication and Concurrency*, Prentice-Hall International, 1989.

28) 尾内理紀夫：『Occam とトランスピュータ』，共立出版，1986.

29) Osborne, R.: Speculative Computation in Multilisp, in 24), pp. 103–137, 1990.

30) Robinson, J. A.: A Machine-Oriented Logic Based on Resolution Principle, *J. ACM*, Vol. 12, No. 1, pp. 23–41, 1965.

31) Saraswat, V. A. and Rinard, M.: Concurrent Constraint Programming (Extended Abstract), in *Conf. Record of the Seventeenth Annual ACM Symp. on Principles of Programming Languages*, ACM, pp. 232–245, 1990.

32) Shapiro, E. Y.: Concurrent Prolog: A Progress Report, *Computer*, Vol. 19, No. 8, pp.

44–58, 1986.

33) Shapiro, E. Y. (ed.) : *Concurrent Prolog : Collected Papers*, Vol. 1–2, The MIT Press, 1987.

34) 柴山悦哉，米澤明憲：並列オブジェクト指向言語 ABCL/1，**bit**, Vol. 20, No. 7–9, 共立出版，1988.

35) Takeda, Y., Nakashima, H., Masuda, K., Chikayama, T. and Taki, K. : A Load Balancing Mechanism for Large Scale Multiprocessor Systems and its Implementation, in *Proc. Int. Conf. on Fifth Generation Computer Systems '88*, ICOT, Tokyo, pp. 978–986, 1988.

36) 上田和紀：並列プログラミング言語，情報処理，Vol. 27, No. 9, pp. 995–1004, 1986.

37) 上田和紀：並列プログラミング言語GHCの設計思想，**bit**, Vol. 19, No. 12, pp. 4–14, 共立出版，1987.

38) Ueda, K. : Guarded Horn Clauses : A Parallel Logic Programming Language with the Concept of a Guard, in *Programming of Future Generation Computers*, Nivat, M. and Fuchi, K. (eds.), North-Holland, pp. 441–456, 1988.

39) Ueda, K. and Furukawa, K. : Transformation Rules for GHC Programs, in *Proc. Int. Conf. on Fifth Generation Computer Systems '88*, ICOT, Tokyo, pp. 582–591, 1988.

40) Ueda, K. and Morita, M. : A New Implementation Technique for Flat GHC, in *Proc. Seventh Int. Conf. on Logic Programming*, The MIT Press, pp. 3–17, 1990.

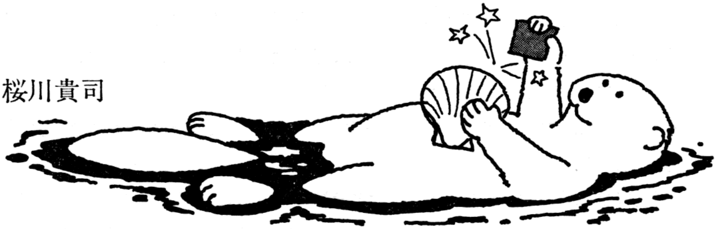
41) Yonezawa, A. and Tokoro, M. (eds.) : *Object-Oriented Concurrent Programming*, The MIT Press, 1987.

42) 大特集：オブジェクト指向プログラミング，情報処理，Vol. 29, No. 4, 1988.

（うえだ かずのり）

第2章　様相論理とテンポラル・プログラミング

桜川貴司



2.1　はじめに

この章の目的は様相論理と時制論理がプログラミング言語にどのように係わってくるかを述べることにある．様相論理といってもたいへんに種類が多く，ここでそのすべてについて述べることは不可能である．また，様相論理の計算機科学への応用も非常に多岐にわたっている．そのため，まず，最も古くからある様相論理である可能性・必然性を扱う様相論理と，計算機科学によく現われる時制論理について述べ，それらのモデルについて触れる．加えて様相のモジュールへの応用の原理を述べ，様相が局所的な述語や関数の定義に使用できることを示す．その後，時制論理の応用例として Prolog の時制論理版である Temporal Prolog，さらに様相をモジュール化に使用した例として RACCO を取り上げる．また，様相論理によって純粋な論理プログラミングの応用範囲が広がる可能性についても述べる．

2.2　様相論理

論理学の本を開いたことのある人で，□とか◇という記号を見たことのある人はどれくらいいるのだろうか．おそらくかなりの割合の人が見たことがあるであろう．

◇や□は様相記号である．そして，様相記号の現われる論理式を取り扱うの

が様相論理である．これらの記号を使って，普通の一階述語論理では直接には表わせないことが扱えるのである．

どのようなことが表わせるかというと，様相論理にもいろいろな種類があるが，そのうちの一つでは，□は必然，◇は可能性を表わす．たとえば，

□(∀X(アメ車(X) ⇒ 左ハンドル(X)))　(1)

という論理式は，アメリカ製の車はすべて左ハンドルであることが必然的であることを表わす．

◇(∀X(日本車(X) ⇒ 左ハンドル(X)))　(2)

こちらのほうは，日本製の車がすべて左ハンドルであるような可能性があることを表わす．(2)は，もし左ハンドルの日本車しか輸入していない国があれば，その国から一歩も外に出たことのない人にとっては，すべての日本車は左ハンドルであることになり，その意味では正しくなる．要するに，非常に単純化して述べると，◇p というのは，pがどこかの国では正しいということである．一方，□p は，pがどこの国でも真であるということである．

一階述語論理では，論理式の真偽値をみる立場，すなわちここでいう国が一つしかなかったのに対し，様相論理では，複数の見方を許しているのである．このことは多世界モデルの節でもう一度述べる．

以上のような論理は，可能性・必然性を取り扱う様相論理で，様相論理の中では最も古く，記号化はもちろんなされていなかったが，Aristotle(アリストテレス）がその創始者である．どうして必然性と可能性がワンセットになっているかというと，両者がある意味でデュアルの関係になっているからである．そうでない可能性はない，ということはすなわちそれが必然的であるということであり，そうでない必然性はない，ということはそうである可能性があるということだからである．具体的な例として(1)の□を除いたものの否定を取ると，

¬(∀X(アメ車(X) ⇒ 左ハンドル(X)))

は，アメリカ製の車は左ハンドルであるとはいえない，という意味である（¬は否定の記号)．すると，

◇¬(∀X(アメ車(X) ⇒ 左ハンドル(X)))

は，アメリカ製の車は左ハンドルであるといえない可能性がある，ということになる．これの否定

$\neg\Diamond\neg$(∀X(アメ車(X) ⇒ 左ハンドル(X)))

は，アメリカ製の車は左ハンドルであるといえない可能性はない，ということである．そういえない可能性はない，ということは，絶対にそうである，ということで，(1)の，アメリカ製の車は左ハンドルであることが必然的である，といっているに等しい．一般化すると，

$\Box = \neg\Diamond\neg$

という関係が成り立つことになる．

様相記号は□◇だけではない．様相論理によっていろいろな様相記号が使われる．たとえば，信念や認識を扱う様相論理では know とか believe という様相が使われる．では，どのような様相が許されるかというと，特に決まっていないようである．要するに，通常の述語論理に現われない命題オペレータ（命題を引数に取り，値も命題であるようなオペレータ．実は，データからデータへの様相というものがあるが，ここでは述べない）を扱うような論理なら様相論理といってもよいようである．様相記号が異なれば，通常その意味が異なるのはいうまでもないが，同じ様相記号を異なる意味に用いる場合もある．次に述べる時制論理では，□◇を必然性・可能性とは異なる意味で用いている．

2.3 時制論理

論理学の記号化の一環として様相論理も記号化，公理化された．いくつもの異なる公理系が提出されたが，時制論理の発端はそのうちの一つに，時間的解釈が見いだされたことによる．

□海は青い (3)

◇空は白い (4)

(3)は可能性・必然性の様相論理では海が青いのは必然的である，という意味になるが，時制論理ではこれからずっと海は青い，という意味になる．また，(4)はいつか空が白くなるときがある，という意味である．

時制論理は計算機と深い係わりがある．計算機のプログラムというのは実行するのに必ずなにがしかの時間がかかる．したがって，計算機の挙動を記述するのに時間の概念が入っている時制論理を用いてみようとするのは自然な発想

である．そのため，様相論理の中では比較的早くからいろいろと計算機に応用されている論理の一つである．

時制論理にもいろいろあり，時間の区間を扱う区間時制論理などもあるが，そのうちの一つに線形時間時制論理がある．この論理は，□◇のほかに○until という二つの様相記号を持っている．そこでは，時間は線形で離散的である．直観的には通常，時間は一次元で，しかも一つの時間が二つに分岐したり，二つの時間が一つに合流したりはしない．また，時間の始めと終りがくっついているような円環的なものでもない．これは最も単純な構造の時間である．また，離散的というのは，時間が連続せず，ディジタル時計のように進んで行く，ということである．まとめると，線形で離散的というのは，時間の構造が整数あるいは自然数と同様になっているということである．

前に戻って，○と until がどういう意味なのかというと，

○晴れ

は次の時刻に「晴れ」であるという意味であり，

(曇り∨晴れ) until 雨

は雨が降るまでは「曇り」か「晴れ」であるという意味になる．それゆえ，時刻の刻みをたとえば一日ごとにとると，

夕焼け ⇒ ○晴れ

は夕焼けになったら次の日は晴れ，という意味だし，

梅雨入り ⇒ じめつく until 梅雨明け

は梅雨入りから梅雨明けまではじめついた天気になる，ということを表わす．

すべての時制論理が離散的な時間構造を持つわけではないが，計算機に使われる時制論理に離散的時間構造を持つのが多い．その理由として，現在のディジタル計算機はクロックに従って動く同期回路からできていることが挙げられる．すなわち計算機においては，元々離散的な時刻しか存在しないのである．また，そのほうが取り扱いやすいのである．

2.4　多世界モデル

この節では，様相論理のモデル論について述べる．論理学でいうモデルとい

うのは，論理式中の変数の変域を集合として定め，関数記号の意味をその集合上の具体的な関数として，また，論理式および述語の一つ一つの真偽値を与えたもののことである．

代表的な様相論理の モデルは 多世界モデル である．一階述語論理の モデルは，世界を一つしか持たない．言いかえると，述語（のグランド・インスタンス：引数のところに変数が現われない形のもの）の真偽値は一意的に定まる．これに対し，時制論理を含めて様相論理の多世界モデルでは，世界によって述語の真偽値が異なってもよい．このことが，様相論理を計算機に応用するときにいろいろな利点となって現われる．実は，真偽値以外にも関数や変数の変域も世界ごとに変わってもよい．

たとえば地球上のそれぞれの国に対して，一つずつ世界を割り当てる．

車は右側通行

という述語は国によって真偽値が異なる．

ところで，世界がお互いにばらばらで，なんの関係もないとすれば様相の登場する理由はない．様相が世界と世界を関係づけるのである．多世界モデルでは，様相というのはほかの世界を覗く窓なのである．

先の例でいうと，それぞれの国がそれぞれの国に代表団を送り，相手の国の様子を知ることができるとしよう．もちろん，代表団を送っていない場合もある．このとき，

□述語

というのが国 A で真であるための条件は，A が代表を送っている国 すべて で述語が真であることである．また，

◇述語

が A で真になるための条件は，A が代表を 送っている国の中で 述語が真であるような国が最低一つはある，ということである．たとえば，

□車は右側通行

が日本で正しいといえるためには，日本が代表を送っている国すべてで車が右側通行であるときに真になる．したがって，たとえば日本が自分自身に代表を送っていたならば，「□車は 右側通行」は，偽となる．もちろん，自分 自身に代表を送っていない場合もあって，この場合には「車は右側通行」の日本にお

ける真偽値には無関係に「□車は右側通行」の真偽値が定まる．一方，

◇車は右側通行

が日本で真になるには，日本が代表を送っている国の中で最低一つの右側通行の国があればよい．

こう書いてくると，□と◇の関係が∀と∃の関係に似ていることに気づかれるであろう．2.2節で述べたように，

□＝￢◇￢

であるが，モデルでの□と◇の意味を考えるとこの関係は，古典論理での，

∀＝￢∃￢

という関係からきているのである．

ある世界（この節の例では世界ではなく国といっていた）a がある世界 b に代表を送っているとき，a から b に到達可能であるといい，aRb と書く．

到達可能性によってモデルはいろいろな構造を持つ．その構造によって様相の入った論理式の真偽値もいろいろと変化する．逆に，到達可能性にいろいろと制限を付けることによって，様相の性質もいろいろと変化する．制限の方法にはいろいろあるが，たとえば，

（1）　aRa

（2）　aRb∧bRc ⇒ aRc

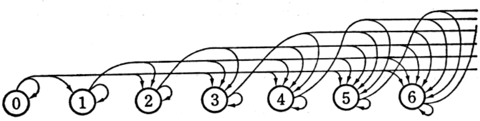
（3）　aRb ⇒ bRa

のようなものである．

時制論理のモデルは，以上の一般のモデルの特殊なものになっている．まず，自然数に1対1対応して世界が可算無限個あるとする．そして，

nRm ⟺ n≦m

となるように到達可能性を定める．図にすると，



というようになる．時制論理では各世界は時刻に対応する．左から若い順に時刻が並んでいるのである．ある時刻で□p が真になるための条件は，この時刻

より右の時刻で必ずpが真になることである．

様相○に対しては，R とは別に到達可能性 R_1 がある．

$$nR_1m \Longleftrightarrow n+1=m$$

すなわち，次の時刻のみが見えるような様相である．時制論理は，複数の到達可能性からなる様相論理なのである．なお，□と◇には，デュアルの関係：□=¬◇¬ があったが，○の場合には○自身が○のデュアルである．

2.5 到達可能性と局所性

局所変数や局所関数，手続きを定義できるプログラミング言語を使ったことのある人なら，局所的定義の有用性を感じたことがあるであろう．といっても，最近（ではなくだいぶ前から）では BASIC でも局所変数を使用できるものがあるため，局所的定義は空気のようになってしまい，かえって有難みを感じないかもしれない．アセンブラでもファイルごとに局所的なラベルを定義できるし，pure Prolog でも変数は節ごとに局所的である．局所的定義はプログラミング言語の世界ではたいへん普遍的である．以上のほかにも，構造体のメンバ名は構造体ごとに局所的である．

一階述語論理では述語は大域的なものであり，同じ名前の述語があちこちで別の意味を持つ，などということはできない．したがって，局所的に述語を使用する，などということは不可能である．しかし，様相論理のモデルでは，世界ごとに述語の意味が異なってもよいため，それを利用すれば局所関数や構造体のメンバ名の局所性と同じようなことが実現できる．

先の国の例を考えてみる．ここで，各国ごとにその国専用の様相を用意し，どの国からもその様相を通して対応する国が見えるものと定める．たとえば，日本には(日本)という様相が対応していて，アメリカにおいても中国においても(日本)pが真になるのはpが日本で成り立つときであるとする．すると，

　(日本) 車は左側通行

は，どの国でも真である．同様にして，

　(アメリカ) 車は左側通行

は偽である．これを見てわかるように，「車は左側通行」という述語は，ある

意味で(日本)や(アメリカ)というモジュールの局所的な述語と見ることができる．もちろん，「車は左側通行」という述語は，(日本)や(アメリカ)という様相を頭に付けなくとも使用することができる．

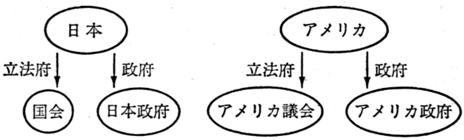
□(車は左側通行 ⇒ 人は右側通行)

というふうに書くと，車が左側通行なら，人は右側通行である，ということが地球上すべての国で正しい，という意味になる．ただし，ここで□は，どの国からもすべての国が見えている様相，とした．また，日本で真であることを書き連ねるのに，

(日本)(**bit** は売れている∧…∧車は左側通行∧…)

というようにして表記することもできる．これはちょうど，日本，というモジュールの中で車は左側通行という述語の意味を定めていることに相当する．Pascal や C で，局所変数や局所関数を定義して使うのと同じような感覚で使うことができるのである．

C のストラクチャや Pascal のレコードのように様相をいくつか積み重ねて使うこともできる．たとえば，日本政府と国会，アメリカ政府，アメリカ議会という世界を用意し，日本から政府という様相で日本政府，立法府という様相で国会，アメリカから政府という様相でアメリカ政府，立法府という様相でアメリカ議会を見られるようにしておく．



すると，

(日本)(政府)借金している

(アメリカ)(政府)借金している

はそれぞれ日本政府，アメリカ政府が借金しているという意味になるし，

(日本)(立法府)荒れている

(アメリカ)(立法府)怒っている

は，それぞれ国会が荒れている，アメリカ連邦議会が怒っているという意味になる．政府や立法府という同じ名前の様相が，世界（この場合は国）によって

違うところを指し，ちょうどレコードやストラクチャのメンバ名と同じ質の働きをしているのがわかる．

以上のような局所的な名前の取扱いには，様相と考える以外にも単なる名前空間の違いとして考える方法もある．どちらがよいかはほとんど好みの問題かもしれないが，筆者としては様相として扱うほうが自然な感じがする．

2.6 純論理プログラミングへ向けて

さて，これ以後は様相論理をプログラミング言語に応用した具体例として，Temporal Prolog と RACCO について述べるが，その前に論理プログラミングへの不満について述べる．

論理プログラミングというと聞こえはいい．確かに，それが有効な場合にはひと味違った味をみせることもある．Prolog を初めて知ったとき，append が両方向に動くのを見てあっと驚いたことを記憶している．しかし，うまくいかないことが多いのも実情である．Prolog でエディタやコンパイラなどを書いても，カットを多用したり，レゾリューションの過程を利用した手続き的なプログラムになったりする．そうなる理由の一つには効率の問題があるが，遅くなっても，あるいはあまり大きな問題が扱えなくてもよいと意識しながらプログラミングしてもうまくいかないことも多い．効率を度外視しても pure Prolog ではうまく書きにくい問題もあるのである．

うまく書けない理由にも場合によりさまざまなものがあるだろうが，その中の一つに pure Prolog の基盤となっている一階述語論理に状態の変化や時間という概念が入っていないことが挙げられる．エディタなどのユーザ・インタフェースが主になるもの，あるいは工場のラインなどを制御する場合には，時間の流れと状態の変化というものがどうしても必要になってくる．一階述語論理にはもともと時間の概念がないため，状態の変化をある意味で無理をしながら扱うことになり，手続き的な動作に依存するプログラムとなってしまうのである．

これの解決策として，時制論理などの時間の概念の初めから入った論理を使うことが挙げられる．もう一つの方法は，なんとか一階述語論理で時間を取り

扱う方法である．具体的には，各述語の引数を一つずつ増やして時間のパラメータを入れればよい．ただし，この方法だと，煩雑になることと，その一方では，不必要かもしれないような複雑なことまで比較的簡単に書けてしまうことが問題である．たとえば，非常に重い恒星が重力崩壊を起こすまでは輝いていることを例に取ると，

輝く until 重力崩壊

∀t2(∀t3(t ≦ t3＜t2 ⇒ ¬ 重力崩壊(t3))

⇒ ∀t3(t ≦ t3＜t2 ⇒ 輝く(t3)))

と比べてみると，上のほうがはるかにわかりやすい．本質をそのまま表わしている．ここで，下の式は，「任意の時刻 t2 に対し，現在の時刻 t から t2 までの間に重力崩壊を起こしていないならば，t から t2 の間輝いている」と読む．また，

輝く(t) ⇒ ∀t1(t＊t ≦ t1 ⇒ 輝いていた(t1))

は，「ある時刻 t に輝いていたとしたら，輝いていたということが時刻 t＊t から以降真になる」ことを記述している．これは，○□◇ until という四つの様相を持つ時制論理では記述できない性質である．この時制論理では時刻がわからないからである．時刻を取ってくる関数 clock があったとすれば記述できるが，

clock＝T∧輝く ⇒ □(clock＝T＊T ⇒ □輝いていた)

となり，こちらのほうがやや長くなり，また技巧的である．もちろん，このような記述を行ないたいこともあるだろうが，それは応用分野によって異なるだろう．むしろ，制限をうまく加えてこういう複雑なことを書きにくくしたほうがよいこともある．また，制限を加えたことにより，論理式の処理や，実行可能性仕様としての実行が可能になるとか，それらの速度が速くなったりする効果も期待される．

2.7 Temporal Prolog

Temporal Prolog は時制論理に基づいた論理型言語である．基礎となっているのは線形時間の時制論理である．特徴は，プログラムは時制論理の論理式で

あること，並列性を記述できること，などである．

使用している様相が増えているため，まずそれらから説明する．

●p	前の時刻で p が真であるとき成り立つ
p since q	q が最後に真になって以来 p がずっと真である
p after q	q が最後に真になってから p が真になったことがある
p for n	p が n 回連続して真になった
◆p	p が真になったことがある
■p	p はこれまでずっと真であった

以上の形の論理式は**条件式**と呼ばれるものである．なお，p や q の位置には別の条件がきてもよい．また，述語自身は条件式である．条件式の否定と条件式を∧(アンド) でいくつか結んだものも条件式である．

次に，

r until p	今から p が成り立つときまで r は真である
r atnext p	これ以後 p が最初に成り立つときに r は真となる
□r	これより r はずっと真である

以上の三つの式は**結果式**と呼ばれる．r にはほかの結果式が入ることができる．また，p には条件式が入る．述語と，p ⇒ r, r∧r1 の形の式も結果式である．Temporal Prolog のプログラムとは，結果式の有限個の集合である．

まず，悪い例から挙げる．

晴 until 春休み ⇒ 山へいく

宝くじが当たる ⇒ ●お金を全部使ってしまう

この二つは駄目な例である．結果式の条件に当てはまっていないことを確認してほしい．

一つ目は，春休みまで晴ならば，今，山へ行く，と読める．このような式は許されない．その理由は，この論理式を真にしようとすると，春休みまで晴であることを予測しなければならないからである．そういうことは，一般には不可能である．

二つ目は，今日宝くじが当たったならば，前の日にお金を全部使ってしまっている，という意味である．これを実行するにはタイムマシンが必要となる．あるいは，お金を全部使えば宝くじが当たるというジンクスでもあれば別であ

るが.

以上の二つの例がいけない例であるということで, 気づかれると思うが, Temporal Prolog では基本的に条件式のところで現在を含めて過去を参照し, 結果式のところで現在を含めて未来の真偽値を操作する. このように制限することによって動かせるものとなっているのである. 実際にはもう少し細かい制限がある[4].

次に規則に沿った例を挙げる.

```
append([ ], X, X)
append(X, Y, Z) ⇒ append([A|X], Y, [A|Z])
●入力(X) ⇒ 出力(X+1)
on ⇒ 点灯 until off
```

上の2行は通常の Prolog とまったく同じものである. Temporal Prolog は pure Prolog を構文上含む.

3行目は数値を入力してそれに1を加えたものを一時刻後に出力するものである.「入力」のように, プログラムによりその真偽値が定まるのではなく, 外部から真偽値が与えられるような述語を**外部述語**という. 逆に, プログラムにより真偽値が決まるものを**内部述語**という.「append」「出力」は内部述語である. 外部述語, 内部述語の区別はプログラムの外で定める. ただし,「append」や「出力」のようにプログラムにより真偽値が左右されるような外部述語はあり得ない.

4行目は, on が押されてから off が押されるまで「点灯」が真になるように働く.

さてここで, 4行目を例に取って, Temporal Prolog と通常の Prolog の比較を行なってみる. 4行目と同じ動作を Prolog で実現するとどのようなものになるであろうか.

```
program :- repeat, on, out(1),
                    repeat, off, out(0), !, program.
repeat.
repeat :- repeat.
```

on, off はそれぞれ入力ポートから読み込んできてボタンが押されていれば

成功し，そうでなければ失敗してバックトラックを起こす述語である．out は出力ポートに引数をそのまま出力する述語である．バックトラック・ループで in が押されるのを待ち，出力を1にし，再びバックトラック・ループで off が押されるのを待ち，出力を0にする．以上をテール・リカージョン・ループで繰り返す．カット・オペレータはテール・リカージョン最適化を行なわせるためのものである．なおテール・リカージョン最適化を行なっていない処理系では，最初のバックトラック・ループでスタックが爆発してしまう．

このプログラムには，論理プログラムの美しさのかけらもない．Prolog の実行手続きを利用した技巧的なプログラムである．もっとも Prolog ハッカーともなると，この手のプログラムに，ある種の美しさを表現できる可能性はあるが，筆者にはそのような能力はない．

上の例は少し極端かもしれない．Prolog にマクロを入れればもっとすっきりした記述が可能かもしれない．しかしながら，たとえそうしたとしても，バックワードに推論を行なうという Prolog 処理系の動作を利用したプログラムである限りは，あくまで Prolog を手続き的な言語として使っているわけである．それでは論理プログラミングとはいえないことはいうまでもない．

2.8 RACCO

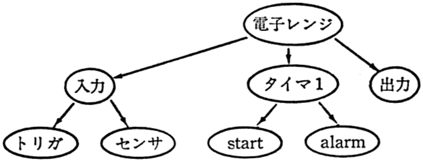
RACCO は時制論理による記述と，様相を用いたモジュール化の機構を基にしたプログラミング言語である．ごく単純にいえば，Temporal Prolog に 2.5 節で述べたモジュール化の機能を取り入れたものである．各モジュールにはそれぞれ独立した複数の様相が対応する．様相には，モジュール本体の様相と，そのモジュールが構成されている部品の様相，それにモジュールがほかのモジュールと通信するための様相とがある．モジュールの宣言には，interface 部と realization 部とがある．interface 部ではそのモジュールの入力述語，すなわち外部から真偽値を操作できる述語と，外部から値を参照できる述語を宣言する．realization 部ではそのモジュールを構成する部品となるモジュールを記述し，また，それらのモジュールがどうつながるかを Temporal Prolog を使って記述する．電子レンジのコントローラを例として挙げる．トリガがかかってか

ら，水蒸気を検出するかタイマが信号を送ってくるまでマグネトロンに通電するようなプログラムである．

```
interface 電子レンジ end interface
realization 電子レンジ with
    入力 is 入力モジュール
    タイマ1 is タイマ
    出力 is マグネトロン
  end with
    (入力)(トリガ) ⟺ (タイマ1)(start)
    (入力)(トリガ)on ⇒
      (出力)on until ((タイマ1)(alarm)on ∨ (入力)(センサ)検出)
end realization
interface 入力モジュール
    state (センサ)検出;
          (トリガ)on;
end interface
interface タイマ
          (start)on;
    state (alarm)on;
end interface
interface マグネトロン
          on;
end interface
```

電子レンジの realization 部では，電子レンジが入力モジュール・タイマ・マグネトロンからなっていて，電子レンジのモジュールの中ではそれぞれ入力・タイマ1・出力という局所名を持っていることを宣言している．これら三つを部品として新たな部品である電子レンジを定義しているのである．その下では，トリガがかかったらタイマをスタートさせると同時に，水蒸気が検出されるか，タイマがアラームを発するまでマグネトロンに通電することを記述している．ここで，(入力)(出力)(タイマ1)(センサ)(トリガ)(start)(alarm)

は様相である．ここで，世界の構造を図示すると以下のようになる．なお，正確には様相は世界から世界を指す関係なのだが，ここでは便宜上世界の位置に様相名を記すことにする．



実はこの図では時刻に関する世界の並びを省略している．正しくは，上のようなものが可算無限個並んでいるわけである．なお以上のうち，

(入力)（トリガ）$\Longleftrightarrow$（タイマ1）(start)

は**パイプ**と呼ばれるもので，(入力)（トリガ）で参照される世界と，(タイマ1)(start) で参照される世界のすべての述語の真偽値が等しくなることを表わす．

ほかのモジュールについては interface 部しかないが，それはこれらのモジュールの機能がハードウェアで用意されるからである．入力モジュールにはトリガとセンサという二つの様相があり，それぞれに検出，on というこのモジュールの外部から真偽値を参照できる述語があることを宣言している．state は，これらの述語の真偽値が外部から変更できないことを意味する．その下のタイマの宣言の (start) on は，state 属性がついていないため，外部から真偽値を変更可能な述語である．すなわち，タイマの入力信号となるような述語である．

2.9 実　現

Temporal Prolog と RACCO の実現方法について簡単に述べておくことにする．

試作した Temporal Prolog の処理系では，すべての結果式を，

$(\bullet)(\neg)p_1 \wedge (\bullet)(\neg)p_2 \cdots \Rightarrow p_6$

という形の論理式に変形してから実行している．ここで，(*) は，括弧内のものが高々一つ現われることを表わす．

たとえば,

入力トリガon ⇒

　　出力 on until タイマ 1on

は,

入力トリガon ⇒ p

●p∧●ㄱ タイマ1on ⇒

p∧ㄱ タイマ1on ⇒ 出力on

という論理式に変換される．ここで，p は新しい述語である．タイマ1が真になるまで状態を記憶する述語として p を加えたわけである．

実行は，各時刻ごとに逐次的に述語の真偽値を求めることによって行なう．過去の履歴と変換後のプログラムから，一つ一つの述語に対して通常の Prolog 処理系によって真偽値を求める．このうち，必要なものについては真偽値を記憶する．具体的には，変換後のプログラムのどこかで●と一緒に現われている述語は，次の時刻での実行の際に真偽値を参照されるので，真偽値を記憶する．その後，次の時刻に移る際にはもう一時刻前の履歴を消去する．変換後のプログラムには高々一つしか●が連なって出てこないので，二時刻以上過去の履歴を参照する必要がないからである．以上を無限に繰り返すわけである．ここで，過去の履歴が消せることがミソで，これができないと記憶領域がパンクしてしまう．

RACCO の実行は，RACCO のプログラムを Temporal Prolog のプログラムに変換することにより行なう．様相により異なる世界に割り当てられている述語それぞれに，一意的な名前を割り当てることにより，様相を消去し，Temporal Prolog のプログラムに変換するわけである．

両者とも，現在の処理系は KCL で書かれており，参考文献 5) に述べてあるエレベータの制御プログラム（約 500 行の RACCO のプログラム）の時刻が1ステップ進むのに約 0.16 秒かかる（データ・ゼネラル MV-10000)．この処理系に用いてある Prolog インタプリタの速度が，約 500 LIPS である．ちなみに，68020 などの Prolog コンパイラで 10KLIPS 以上のものもある．Temporal Prolog のコンパイラをきちんと作ったときに以上の速度比がそのまま保存されるとすれば，時刻1ステップの実行に約 8 m 秒かかることになる．

2.10 まとめと参考文献案内

以上，様相論理とそのプログラミング言語への応用を解説するとともに，様相が時間の概念の必要な記述や，モジュール化などに使用できることを述べた．

Temporal Prolog のさらに詳しい解説と，その形式的意味を述べたものに 4) がある．また，RACCO のさらに詳しい解説には 5) がある．

時制論理を用いたハードウェア記述やプログラミング言語の文献としては，1)，3)，6) などがある．1) は論理変数の値が時刻とともに変化する論理を用いている．6) は区間時制論理を用いている．また，2) には，Prolog を，時制論理とは限らない一般の様相を用いて拡張する方法について述べてある．

ここで述べたものは主に論理プログラムへの応用であったが，その可能性を理解していただけたら幸いである．

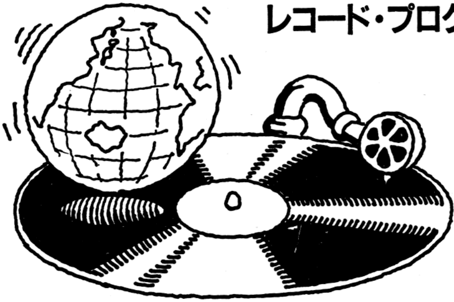
参考文献

1) 青柳龍也，藤田昌宏，元岡　達：Tokio かける言語，*Proceedings of the Logic Programming Conference '85*, 8.1.
2) Farinas Del Cerro, L.: MOLOG: A System that extends Prolog with Modal Logic, *New Generation Computing*, 4, pp. 35–50, 1986.
3) Moszkowski, B.: Executing Temporal Logic Programs, Technical Report No. 71, Univ. of Cambridge Computer Laboratory, 1985.
4) 桜川貴司：Temporal Prolog，コンピュータ・ソフトウェア，Vol. 4, No. 3, 岩波書店，1987.
5) 桜川貴司，竹中一起，中島玲二，新出尚之，服部隆志：RACCO：実時間プロセス制御システムのモデル記述のための様相論理プログラミング言語，コンピュータ・ソフトウェア，Vol. 5, No. 3, 岩波書店，1988.
6) 米崎直樹，新　淳，蓬莱尚幸：時間論理プログラミング言語 Templog，日本ソフトウェア科学会第1回大会論文集.

（さくらがわ たかし）

第3章

レコード・プログラミング



横田一正

3.1 はじめに

“レコード・プログラミング”という言葉は，あまり聞き慣れない言葉かもしれない．本章では，この言葉を“レコード構造を基にしたプログラミング言語”という意味で使用しているが，そのままだと標題として長すぎるので，単に“レコード・プログラミング”とした．よく似た言葉として“データベース・プログラミング言語（DBPL；Database Programming Language）”がある．これは，データベース言語とプログラミング言語を統一しようとするもので，これも新しい分野として活発な研究が行なわれている[17,20]．広義には，レコード・プログラミングもこの中に入れることができよう．しかし最近のサーベイ[14]にも見られるように，この言葉には多くのアプローチが混在している．たとえば，SQL のような古いデータベース言語と Pascal のような古いプログラミング言語を“プラグマティックに”統合することも含んでおり，このイメージが払拭されない限り，DBPL という言葉は本書の表題『続　新しいプログラミング・パラダイム』に似つかわしなくいと考えた．また本章での対象が必ずしもデータベース言語だけを対象にしていないことから，あえて“レコード・プログラミング”という言葉を使った．

言葉の問題はさておき，まず，なぜレコード・プログラミングなのか，を考えてみよう．プログラミング言語とは，実世界の問題を計算機上で解決するための記述言語であり，そのためにアルゴリズムのほかにデータの記述も当然で

きるようになっている．データとは，実世界の“もの”（“オブジェクト”あるいは“エンティティ”（実体）といってよいかもしれない）を記号世界で表現したものとか，記号世界で新たに作り出したものとか，あるいはプログラム自身であったりする．それらは，表現と操作のパターンによって“基本データ型”として，いくつかに分類される[8]：

● 原子データ型

　論理型，文字型，整数型，実数型，…

● 構造データ型

　配列型，直積型，直和型，集合型，列型，再帰型

● 参照データ型

　ポインタ

原子データ型は個々の値の性質を，構造データ型は構造の鋳型を，参照データ型はデータ間の関係を示している．レコード型は上記の直積型に対応するが，本章で“レコード構造”というのは，内容としてはさらに配列型，集合型，再帰型を含んでいる．したがってかなり一般的なデータを対象としているといってよい．

このようなデータ構造は，これまでも多くのプログラミング言語で考えられてきたし，実際にその機能を持っている．しかし問題は，そのデータ構造が非常に一般的な形式であるために，現実のさまざまな問題領域のデータ自身が持っている意味を反映していないことである．問題領域での意味の扱いはユーザ・プログラムの責任とするのが“ふつう”とされている．ところが，知識処理の急速な展開とともに，各問題領域の対象をより直接的に表現でき，かつその意味を反映した“高いレベル”のプログラミング言語の必要性が強くなっており，すでにその流れに沿った多くの言語が提案されている．その中でレコード構造を意識した言語を“レコード・プログラミング”と総称した．述語と関数子によってレコード構造を表現する Prolog もこの範疇に入ると考えられるが，本章では，よりレコード構造に着目しているものを対象としたい．

本章では，ほかの章とは少し趣を変え，まずレコード構造の表現に関する考察から始める．そしてレコード・プログラミングで最も影響力を持っている，H. Aït-Kaci の ψ-項とそれに基づいた言語 KBL と LOGIN について，例を

中心に説明する．そして最後に，レコード・プログラミングの応用の一つとして，データベース分野での現状を簡単に紹介することにする．詳細はいずれも参考文献を参照されたい．

3.2 レコードと述語の表現

まずレコードの定義を考えてみよう．Pascal 風の記法では次のように定義する：

```
type T =record
        a1: T1;
          ⋮
        an: Tn;
end.
```

これはレコードの型Tを定義するもので，$a_1, \cdots, a_n$ は属性名，$T_1, \cdots, T_n$ はすでに定義されている型（あるいは原子データ型）である．これは，属性の集合から型の集合への関数と見ることができるので，$T(a_i)=T_i$ という演算が定義される．これは $T.a_i=T_i$ とも書かれる．

ここでは一般的に，(**record**…**end** の代わりの）組構成子（tuple constructor）"［"，"］" と集合構成子（set constructor）"｛"，"｝" によって構成される "オブジェクト" を考え，そのサブクラスとしてレコードを考えよう．話の順序としては型ではなく，オブジェクト自体の定義から始める．まず原子オブジェクトの集合 O と属性名の集合 A が与えられているとする．するとオブジェクトは次のように定義される：

（1） $o \in O$ ならば o はオブジェクトである．

（2） $o_1, \cdots, o_n$ がオブジェクトなら $\{o_1, \cdots, o_n\}$ はオブジェクトである．これを**集合オブジェクト**という．

（3） $a_1, \cdots, a_n \in A$ が異なる属性名で，$o_1, \cdots, o_n$ がオブジェクトなら $[a_1 \rightarrow o_1, \cdots, a_n \rightarrow o_n]$ はオブジェクトである．これを**組オブジェクト**という．

これまでレコードといっていたのは，一般的には組オブジェクトであり，現実的にはそれに，構造上の制限などを付加したものと考えることができる．組オ

ブジェクト $O=[a_1 \to o_1, \cdots, a_n \to o_n]$ に対しては，上記と同じく $O.a_i=o_i$ という演算が定義される．型は（いくつかの流儀があるが）これらオブジェクトの抽象構造として定義することができる．詳細な定義は省略するが，たとえば，人のレコード型は

人＝[名前 → 文字列,
　　生れ → [出身地 → 文字列,
　　　　　　年月日 → [年 → 整数,
　　　　　　　　　　　　月 → 整数,
　　　　　　　　　　　　日 → 整数]],
　　趣味 → {文字列}]

のように定義できる．さらにオブジェクトと型を組み合わせて，'オブジェクト：型' と書くと，

[名前 → "山田"：文字列,
　生れ → [出身地 → "東京"：文字列,
　　　　　年月日 → [年 → 1960：整数,
　　　　　　　　　　　月 → 11：整数,
　　　　　　　　　　　日 → 13：整数]：T_1]：T_2,
　趣味 → {"野球", "音楽"}：{文字列}]：T_3

と型付きオブジェクトを簡単に再定義できる．ただし T_1, T_2, T_3 はそれぞれ

T_1＝[年 → 整数, 月 → 整数, 日 → 整数]

T_2＝[出身地 → 文字列, 年月日 → T_1]

T_3＝[名前→文字列, 生れ → T_2, 趣味 → {文字列}]

と定義される型である．また再帰的な型の定義にも簡単に拡張できる．このような型は必ずしもすべてを明記しなくても，一部の型から残りを容易に推論できる[9]．

このようなレコード表現に対して，述語論理（あるいは Prolog）での述語表現を考えてみよう．述語は集合構成子を持っていないので，下記の議論は組構成子の場合に限ることにする．もともと述語自体がそもそもオブジェクト間の関係を表現しているので，レコードの一種ともいえる．たとえば，『太郎は花子が好き』とか『次郎と桜子は3月にハワイで結婚し，新婚旅行は欧州であ

る』は，述語では，それぞれ

好き（“太郎”,“花子”）

結婚（“次郎”,“桜子”,“3月”,“ハワイ”,“欧州”）

と表現できる．‘好き’や‘結婚’という述語名やその引数の個数やそれらの役割は，記述しようとしている問題に応じて便利なように決めることができる．しかし，この表記の欠点はすぐ明らかになる：

● 引数の位置の問題

述語では引数の位置（順序）が役割を示しているが，これは対象を表現するのに不便であろう．‘結婚（…）’のように引数が増えてくると，よほどの記憶力の持ち主でないと，すぐ忘れてしまう．

● 引数の個数の問題

Prolog では述語名と引数の対で述語を識別するように，引数の個数は述語表現にとっては本質的な役割を果たしている．ところが，情報の一部しか得られていないとき，たとえば

結婚（“次郎”,“桜子”,＿,“ハワイ”,＿）

のように，下線または変数で情報の欠落を表現しなければならない．

これらを改善するために，述語記法ではなく，上記のレコード記法を用いることにしよう．すると上の述語は，たとえば

［関係 → 好き, 主語 → “太郎”, 目的語 → “花子”］

［関係 → 結婚, 夫 → “次郎”, 妻 → “桜子”,

日程 → “3月”, 式場 → “ハワイ”, “新婚旅行 → “欧州”］

のように表現できる．あるいは‘好き’と‘結婚’を型とすれば，

［主語 → “太郎”, 目的語 → “花子”］：好き

［夫 → “次郎”, 妻 → “桜子”, 日程 → “3月”,

式場 → “ハワイ”, 新婚旅行 → “欧州”］：結婚

とそれぞれ表現できる（この型表現の便利さは後述するが，たとえば，‘新婚 is_a 結婚’とか，‘再婚 is_a 結婚’のように，型階層を導入しやすいことである）．この表現では，属性名が付随する値の役割を示しているので，引数（項）の順序は無関係とすることができる．また部分情報については，

［関係 → 結婚, 夫 → “次郎”, 妻 → “桜子”, 式場 → “ハワイ”］

のように判明しているところだけを記述すればよい．述語記法とレコード記法を一般的に対比すれば，$p(t_1, \cdots, t_n)$ という述語に対して，層性名 \$0, \$1, …, \$$n$ を導入することによって，

$$p(t_1, \cdots, t_n) \Rightarrow [\$0 \rightarrow p, \$1 \rightarrow t_1, \cdots, \$n \rightarrow t_n]$$

$$\text{または} \qquad [\$1 \rightarrow t_1, \cdots, \$_n \rightarrow t_n] : p$$

と変換することができる．つまり，このレコード記法は通常の述語表現の拡張になっており，しかも，柔軟である．変数を含んだレコードへの拡張も容易に行なうことができる．したがって Prolog をそのままレコード構造へ書き写せることが，直観的にわかるだろう．

このようなレコード記法にこだわる理由をまとめると次のようになる：

- 述語記法に比べて“高いレベル”の表現，つまり実世界により近い表現になっていること．
- さらに複雑な構造，たとえば集合構成子によって集合のような構造も導入しやすいこと．
- （次節以降で説明するが）継承関係などの各問題領域の意味を導入しやすいこと．

したがってレコード・プログラミングでの問題は，このようなレコード構造への，各問題領域の意味の導入と，それを単に（Prolog のような）“低いレベル”の言語に変換するのではなく，そのままの表現を基にした“きれいな”(論理的基礎を持った）プログラミング言語を構築することである．

3.3　レコード構造と ψ-項

レコード構造を基にしたデータ表現を，非常に美しく形式化し，プログラミング言語に応用したのは，Hassan Aït-Kaci である．1984年にそれをペンシルバニア大学の学位論文

“A Lattice Theoretic Approach to Computation Based on a Calculus of Partially Ordered Type Structures”

(『半順序の型構造の論理に基づく計算への束論的アプローチ』)

にまとめあげ，整理したものを参考文献 12) に掲載した．彼のもともとの動機

は型の継承を形式化することにあったが，この形式的枠組みは一般のレコード構造に適用できるので，この研究の影響力は非常に大きく，データベースや論理型言語の分野を中心に，関連した研究が活発に続けられている．Aït-Kaci の論文は以下のようにまとめることができる：

（1） 型（構造）をレコード構造として表現する．これを ψ-項（Psi-term）という．これは先に指摘した述語表現の欠点を克服するためである．

（2） 原子型間の包摂関係（半順序）を ψ-項間の包摂関係に拡張し，それから束を構成する．

（3） ψ-項間の交わり演算（単一化）のアルゴリズムを与える．

（4） ψ-項に集合表現を導入した ε-項に拡張する．

（5） （2），（3），（4）と項書換え[2)]に基づいたプログラミング言語 KBL を定義する．

これが学位論文の内容であるが，Aït-Kaci は学位論文取得後，アメリカの第五世代コンピュータ・プロジェクトの一つである MCC に行き，Prolog に ψ-項を組み込んだ LOGIN という言語，さらに ψ-項と論理型言語と関数型言語の統合をねらった LIFE という言語の研究を精力的に行なった．本節では，ψ-項とその性質，そしてそれを基にした KBL と LOGIN の，例による簡単な紹介にとどめる．

3.3.1 ψ-項の定義

まず‘人’を定義する例から始めよう．ψ-項では次のように表現される：

```
人(識別 → 名前,
  生れ → 年月日(年 → 整数,
               月 → 整数,
               日 → 整数),
  父親 → 人)
```

これは簡単な ψ-項である．前節のレコードと対応させれば，これは型の定義に対応している．‘人’，‘名前’，‘年月日’，‘整数’が型で，‘識別’，‘生れ’，‘年’，‘月’，‘日’，‘父親’が属性名である．前節のレコード記法と対応させれば，値のないレコードとなっている（再帰的な定義になっていることに注意）：

[識別 → 名前,
生れ →[年 → 整数,
月 → 整数,
日 → 整数]：年月日,
父親 → 人]：人

ここでは，ψ-項に似せ，組構成子によって構成されたオブジェクト（型）の場合には，型を左側に付けた記法を使用する：

人[識別 → 名前,
生れ → 年月日[年 → 整数,
月 → 整数,
日 → 整数],
父親 → 人]

Aït-Kaci はさらに，述語表現の分析からタグを付加する．つまり述語中の変数の役割を分析し，

- 何を代入してもよいという“ワイルド・カード”の役割
- 同じ変数には同じものが代入されねばならないという等値制約の役割

の二つを分離した．そして，前者のためには最も大きな型（⊤）を使用し，後者のためにはタグと呼ぶ等値制約を導入した．（3.3.2 項で説明するが，⊤と任意の型 γ との下限は γ になるので，⊤は代入に関しては変数の役割を代行している.）たとえば

人[識別 → 名前[姓 → X：文字列,
名 → 文字列],
住所 → ⊤,
父親 → 人[識別 → 名前[姓 → X]]]

を考えよう．‘：’の前の X は変数でも値でなくタグである．タグと型は容易に識別できることを前提にして，ほかの部分では区切り記号‘：’も省略している．‘住所’の型が‘⊤’であるのは，それは何でも（‘文字列’でも‘整数’でも‘年月日’でも）よいことを示している．タグ“X”は，この例では，‘人’の‘姓’は‘父親’と同じ‘姓’でなければならないという等値制約を表現している．しかしこれは無制限に付加できるものではない．たとえば

人[識別 → 名前[姓 → X：文字列,
　　　　　　　　名 → 文字列],
　年齢 → X：整数]

のように，‘文字列’ と ‘整数’ のような排他的な型に同一のタグを付加することは当然禁止される．

また注意すべきことは，この表現は無限の構造を簡単に表現できることである．前述の ‘人’-‘父親’ 関係もその例となっている．また，次の例

X：人[識別 → 名前[姓 → 文字列,
　　　　　　　　　名 → 文字列],
　　　恋人 → 人[恋人 → X]]

は，相思相愛の恋人のいる人を表現している．しかし一般的な無限構造を許すと，望むべきよい性質（束演算の停止性）の障害となるので，制限を加える．つまり無限は同一のパターンの繰返しに限る，という制限である．上の例はそれを満足している．

それでは一般的な ψ-項の定義を考えてみよう．上の例で，人を指す属性名と，姓の下のタグ X 以外は省略されていると考え，正式には以下のように書く：

e → X_1：人[識別 → X_2：名前[姓 → X：文字列,
　　　　　　　　　　　　　　名 → X_3：文字列],
　　　　住所 → X_4：⊤,
　　　　父親 → X_5：人[識別 → X_6：名前[姓 → X：文字列]],
　　　　恋人 → X_7：人[恋人 → X_1]]

つまり先頭の e は特殊な空属性であり，明示されていないタグはそこにしか現

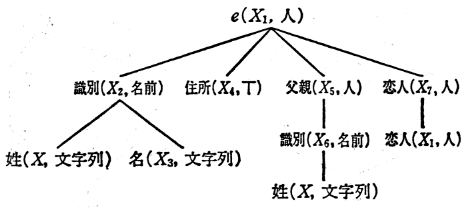


図 3.1 ψ-項の定義

われないものである．したがって，これは図3.1のように，属性名からなる木の各ノードにタグと型を付加していると考えられる．各ノードは，そこに至る属性名列によって識別されるので，ψ-項は，属性名列の集合（つまり層性名から構成される木領域）Δ と，Δ からタグ集合への関数 γ，そして Δ から型集合への関数 ψ の三つ組 (Δ, ψ, γ) として定義できる．これに上記のような制限を付加したのが ψ-項である．

この ψ-項の特徴的な点は，“太郎”，“1988” など通常は値あるいは個体として考えられるものも，型の一種として扱うことである．つまり，個々の値も上記のように構成された ψ-項もすべて型であり，この型の世界を対象としている．この考えは，「すべてがオブジェクト」と考えるオブジェクト指向の考えとも調和しており，後述するが，その構造的側面に主として着目しているオブジェクト指向データベースの形式化にも大いに適用することができる．

3.3.2 型の半順序と束

次に考えねばならないのは，問題領域の意味の記述，つまりこの場合は型の継承関係である．そのために，型に半順序を導入する．つまり，‘大人 is_a 人’ を ‘大人 $\preceq$ 人’ という順序に対応させるのである．型の意味としては集合を対応させ，この順序には集合の包含関係を対応させる．

まず原子型の間に半順序が与えられると，その順序を保存した形で束に変換することができる．たとえば，型として {単位, 点数, 優, 良, 可, 不可, 落第点, ⊥, ⊤} を，そして半順序として

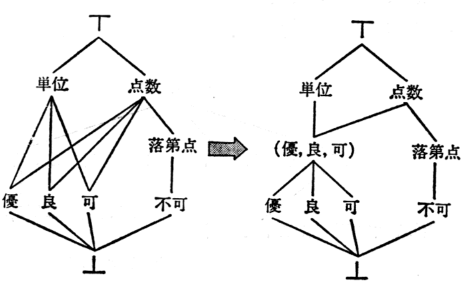


図 3.2 半順序から作られる束

優⪯単位, 良⪯単位, 可⪯単位, 不可⪯落第点,

優⪯点数, 良⪯点数, 可⪯点数, 落第点⪯点数

を与えると，**図 3.2** の左のように，‘優’，‘良’，‘可’ の上限は一意的に決まらないので束ではない．そこで‘(優, 良, 可)’という新しい型を導入し，図 3.2 の右のようにする．これは束になっている．つまり

合格点＝(優, 良, 可)

という新たな型を定義し，半順序として

合格点⪯点数, 合格点⪯単位,

優⪯合格点, 良⪯合格点, 可⪯合格点

を導入するのと同じである．

このように原子型の集合が束になると，それらから作られる ψ-項も束にすることができる．これが ψ-項の中心となる性質である．通常の一階述語の世界を考えたとき，そこで構成される項の集合（に⊤と⊥を加えたもの）は単一化（unification）と一般化（generalization）を，それぞれ下限と上限とする束になっているが，それを比べると，ψ-項の場合は，述語記号と関数子が型としてさらに階層をなしていると考えられる．少し複雑な例で考えよう．まず，原子型の階層として**図 3.3** を与える．そしてその上で構成される二つの ψ-項

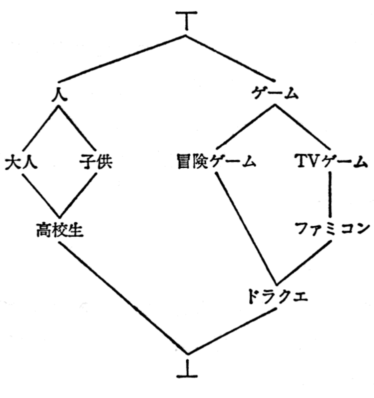


図 3.3 ゲームの束構造

子供[知る → X : 人[知る → ファミコン,
　　　　　　　　嫌い → Y : TV ゲーム],
　　嫌い → 子供[知る → Y,
　　　　　　　好き → ドラクエ],
　　好き → X]
大人[知る → 大人[知る → 冒険],
　　嫌い → 人[知る → X : TV ゲーム,
　　　　　　好き → X]]

を考えよう．‘大人’ と ‘子供’ などを述語表現の述語記号 あるいは 関数子と考えれば，単一化は失敗するが，ψ-項の束を考えれば，それらの上限（一般化）と下限（単一化）はそれぞれ次のようになる：

人[知る → 人[知る → ゲーム],
　嫌い → 人[知る → TV ゲーム,
　　　　　好き → TV ゲーム]]
高校生[知る → X : 大人[知る → ドラクエ,
　　　　　　　　　嫌い → Y : ドラクエ],
　　嫌い → 子供[知る → Y,
　　　　　　　好き → Y],
　　好き → X]

原子型の束演算のほかに，属性名とタグについての束演算が付加されており，一階項に比べて 拡張が 行なわれている．また，下限の ‘高校生’ には，‘大人’ と ‘子供’ の属性が単一化とともに継承されているのもわかるだろう．つまり，ψ-項を基にしたプログラミング言語は，型の階層に従った継承を，このように行なうことが期待できる．

3.3.3 KBL と LOGIN

このような ψ-項から，Aït-Kaci は項書換えシステム[2)]を基にしたプログラミング言語 KBL を作った．その前に集合構成子を使って ψ-項を拡張した ε-項を説明しよう．たとえば

人[性別 → {男, 女},

父親 → Y：人[性別 → 男],

母親 → Z：人[性別 → 女],

親 → {Y, Z}]

は ε-項である．これは集合を持たない ψ-項の集合へと展開される．この場合は四つの ψ-項である．さて KBL は以下のような書換え規則からなる：

$$
\begin{aligned}
&\mathit{list} = \{\mathit{nil}, \mathit{cons}\};\\
&\mathit{append} = \{\mathit{append0}, \mathit{append1}\};\\
&\mathit{append0} = [\mathit{front} \to \mathit{nil},\\
&\qquad \mathit{back} \to X : \mathit{list},\\
&\qquad \mathit{whole} \to X];\\
&\mathit{append1} = [\mathit{front} \to \mathit{cons}[\mathit{head} \to X,\\
&\qquad\qquad \mathit{tail} \to Y],\\
&\qquad \mathit{back} \to Z : \mathit{list},\\
&\qquad \mathit{whole} \to \mathit{cons}[\mathit{head} \to X,\\
&\qquad\qquad \mathit{tail} \to U],\\
&\qquad \mathit{patch} \to \mathit{append}[\mathit{front} \to Y,\\
&\qquad\qquad \mathit{back} \to Z,\\
&\qquad\qquad \mathit{whole} \to U]].
\end{aligned}
$$

これは，リストの *append* のプログラムである．*list*, *nil*, *cons*, *append*, *append*0, *append*1 が型であり，定義から型間の半順序として，$\mathit{nil} \le \mathit{list}$, $\mathit{cons} \le \mathit{list}$, $\mathit{append0} \le \mathit{append}$, $\mathit{append1} \le \mathit{append}$ を考える．それぞれの等式は，“＝”の左の型を右の型に書き換えることを示している．具体的に

$$
\begin{aligned}
&\mathit{append}[\mathit{front} \to \mathit{cons}[\mathit{head} \to 1,\\
&\qquad\qquad \mathit{tail} \to \mathit{cons}[\mathit{head} \to 2,\\
&\qquad\qquad\qquad \mathit{tail} \to \mathit{nil}]],\\
&\qquad \mathit{back} \to \mathit{cons}[\mathit{head} \to 3,\\
&\qquad\qquad \mathit{tail} \to \mathit{nil}]]
\end{aligned}
$$

という型が与えられると，型の単一化と書換えによってプログラムが実行さ

れ，*whole* の型が

$$cons[head \to 1,$$
$$tail \to cons[head \to 2,$$
$$tail \to cons[head \to 3,$$
$$tail \to nil]]]$$

となって停止する（詳細は参考文献12）を参照されたい）．これが実行結果になっている．このプログラムに対応させて，同じ *append* を Prolog で書くと，

$$append(X,Y,Z) :\text{-}\ append0(X,Y,Z).$$
$$append(X,Y,Z) :\text{-}\ append1(X,Y,Z).$$
$$append0(L,X,X) :\text{-}\ L = nil.$$
$$append1(L1,Z,L2) :\text{-}$$
$$L1 = cons(X,Y), L2 = cons(X,U), append(Y,Z,U).$$

と書くことができる．Prolog のリダクション[5]を考えれば，両者の関係は明らかでろう．KBL は，ψ-項のプログラミング言語としてはプロトタイプという性格が強く，見やすさは劣っているが，その可能性は十分に示しているように思える．

LOGIN は，MCC の論理型言語の環境で提案された言語で，Prolog の通常の項の代わりに，ψ-項を使っている[13]．つまり，述語の各項の単一化の代わりに ψ-項の単一化を行なう．これは ψ-項の本来の考えからすれば，妥協の産物といえるかもしれない．型の階層が図 3.4 のようになっていたとしよう．そして

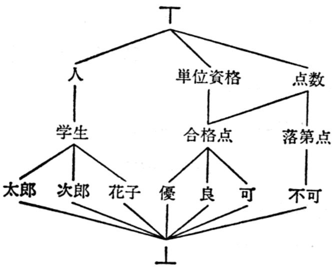


図 3.4　単位と学生の束構造

次の LOGIN プログラムを考えよう：

like(*X* : 人,*X*).

like(太郎, 花子).

like(人, 単位資格).

got(太郎, 不可).

got(次郎, 不可).

got(花子, 優).

happy(*X* : 人):- *like*(*X*,*Y*), *got*(*X*,*Y*).

happy(*X* : 学生):- *like*(*X*,*Y*), *got*(*Y*, 単位).

ゴールが ?-*happy*(*X*：人) のとき，Prolog と同じく導出が行なわれる．項は ψ-項であるので，たとえば‘*X*：人’と‘太郎’の単一化は，単一化失敗ではなく，‘*X*：太郎’が得られる．Prolog のリダクションと同様に進み，ゴールの型‘*X*：人’が，導出によって，‘*X*：花子’あるいは‘*X*：太郎’になって成功するのは容易に確認できるだろう．(この例は非常に単純であるが，3.3.2 項の‘大人’と‘子供’の例を代わりに考えれば，単一化と同時に継承の実行が理解できるだろう．)

3.4 応用——データベースの視点から

データベース分野では，データモデルに対応して従来から多くのデータベース言語が提案されてきた．階層型モデル，網型モデル，関係モデルのデータが，すべてレコード表現可能なことも以前から指摘されている[21]とおり，データベース言語とレコード・プログラミング言語は深く関係している．この節では，この中で，演繹データベースの拡張としてのレコード・プログラミング言語を中心に説明する．そのためにまず演繹データベースの説明から始めよう．

3.4.1 演繹データベース

レコード構造の最も簡単な，平坦な場合を考えてみよう．これを基にしたデ

ータベースは，関係データベースとして知られている．関係に含まれる各レコードのことを**組**（tuple）という．演繹データベースと述語論理の親和性を出発点としており，下記のような対応関係を持っている：

組 ⟺ 述語

データベース ⟺ 理論

問合せ処理 ⟺ 定理証明

演繹データベースには，事実（ファクト）のほかにルールも含まれており，それによって，データベースに含意されている事実を推論することができる．

たとえば，参考文献3) の例を考えよう．ノード a, b, c, d があり，その間が**図3.5**のようなアークで結ばれていたとする．これを関係と述語の集合（理論）でそれぞれ表現すると**図3.6**のようになる．

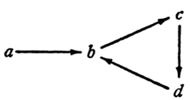


図 3.5　ノードとアーク

関係 *ARC*：

from	to
a	b
b	c
c	d
d	b

述語（理論 *ARC*）：

アーク(a, b)
アーク(b, c)
アーク(c, d)
アーク(d, b)

図 3.6　関係と理論

理論 *ARC* から論理式 p が証明できることを $ARC \vdash p$ と書くと，「ノード b にアークが伸びているノードを求めよ」という問合せの答えは

$\{X \mid ARC \vdash$ アーク$(X, b)\}$

と書くことができる．したがって上記の関係と理論との対応関係は容易に理解できるだろう．演繹データベースでは，上記のような事実のほかにルールを追加することによって，関係データベースを拡張している．たとえば，あるノードからほかのノードへの「パス」（到達可能性）は以下のように定義できる：

パス(X, Y) :- アーク(X, Y).

パス(X, Y) :- アーク(X, Z), パス(Z, Y).

すると，演繹データベースでは「ノード b から到達可能なノードを求めよ」という推移閉包の問合せが可能になる．演繹データベースはこのように，従来のデータベースに対応する事実の集合のほかに，ルールの集合を考え，それらもデータベースの一部と考える．格納するルールの形を確定 Horn 節に限れば，

これは Prolog のプログラムそのものでもある．そこで

演繹データベース $\Longleftrightarrow$ 一階理論 $\Longleftrightarrow$ 論理プログラム

という対応関係が成り立っている．

データベースとプログラムはこのような対応関係を持っているが，評価方法がそれぞれ異なっていることに注意する必要がある．たとえばデータベースは量の問題に対処するために，集合操作を基本にしているが，Prolog のようなプログラミング言語は組単位の実行機構しか持っていない．そこでこの解消のために，LDL (Logic Based Data Language)[16]は次のような記述を許している：

$p(X, <Y>)$:- $q(X, Y)$.

<…>は集合のグルーピング（set-of）の機能を表わしており，この例ではボディを満足する Y のすべてが集合の形で$<Y>$に求められる．この set-of はかなりの表現力を持っており，集合の集合はもちろん，空集合を利用することによって否定も表現することができる．したがって，何の制限もないと Russel（ラッセル）のパラドックスも容易に記述できるので，**層状化** (stratification) という制限が必要となる．しかしこの枠組みで，複雑な構造のデータをより直接的に表現するのはできないので，さらにこの中に ψ-項を組み込むことも検討されている．

3.4.2 レコード・プログラミングとデータベース

演繹データベースの拡張がさまざまな視点から研究されている．それらの研究は以下のように分類できる[10]：

（1） 論理的拡張

否定，選言，存在限量子，空値，確信度などの導入

（2） データモデル的拡張

(a) 構造的拡張

非正規関係，複合オブジェクトなどの構造データの導入

(b) 手続き的拡張

データと手続きのカプセル化

(c) オブジェクト識別性の導入

識別子によるオブジェクトの直接的表現

（3） 計算モデル的拡張

(a) 制約パラダイム[1)]

オブジェクトを制約の集合と考え，制約論理言語の導出，制約評価系(constraint solver）による評価によって問合せを処理する．

(b) オブジェクト指向パラダイム[4)]

各オブジェクトを自律的問合せ処理系と考え，メッセージ・パッシングとオブジェクトの内部処理によって問合せを処理する．

（1）は演繹データベースの論理的な表現能力の拡張であり，演繹データベースの研究の当初から活発に研究が行なわれており，最近では人工知能での非単調推論などとの関係も注目されている．

（2）は実世界の対象をいかに自然にモデリングするかという視点からの拡張で，単にデータ構造の拡張だけではなく，オブジェクト指向[4)]の（受動的オブジェクトの）考えに沿った拡張が盛んになっている．

（3）はデータベースでの計算過程，つまり問合せ処理をいかに自然にモデリングするかという視点から研究されている．

これらはさまざまなレベルで互いに組合せが可能で，この枠組みによって多種のデータベースを考えることができる．

その中で，本章でのテーマであるレコード・プログラミングは，3.2節で検討したように，（2）の拡張と深く関係している．（2）(a)の構造的拡張に焦点を当てたものに，

- Calculus for Complex Objects，F. Bancilhon ほか，1986[15)]
- COL，S. Abiteboul ほか，1987[11)]
- CRL，K. Yokota，1987[28)]
- C-Logic，W. Chen ほか，1989[18)]

などがあり，さらに，「オブジェクト指向」を表に出し，（2）の全体を実現しようとするものに

- O-Logic，D. Maier，1986[26)]；改訂版，M. Kifer ほか，1989[22)]
- F-Logic，M. Kifer ほか，1989[23)]，1990[24)]
- DOT，塚本ほか，1989[7)]
- Juan，K. Yokota，1990[30)]（現在は，Quixote, H. Yasukawa and K. Yokota, 1990）

などである．発表年を見てわかるとおり，これらの研究が盛んに行なわれるようになったのは，ごく最近のことである．単なる構造データの拡張から，オブジェクト指向（受動的オブジェクト）概念の導入へと向かった理由には，このような拡張が当初の目的（より自然なデータ・モデリング）にかなっていたからである．この方向，つまり演繹データベースとオブジェクト指向データベース[6]の統合を目指した国際会議も開催されている：

The First International Conference on Deductive and Object-Oriented Databases (DOOD89)[25]，京都，1989年12月4日～6日

主催は情報処理学会と京都高度技術研究所（ASTEM RI/Kyoto）で，後援は，日米欧の次世代コンピュータの研究機関である ICOT, ECRC, INRIA, MCC である．次世代データベースの核としてのレコード・プログラミングの重要性と，今後の新しい発展の必要性が活発に議論された．

3.4.3 いくつかの例

それでは具体例をいくつか見てみよう[10]．ただし，記述の統一性から，元の記法を修正していることをお断りしておく．

O-論理[22]では，下記のように属性値に集合の記述を許している：

花子：職員[名前 → “花子”：文字列,
　　　　　年齢 → 30：整数,
　　　　　趣味 → {チェス：ゲーム, テニス：ゲーム},
　　　　　職場 → CS_2：学科[学科名 → “情報工学科”：文字列]]

ここで，‘職員’，‘文字列’，‘整数’，‘ゲーム’，‘学科’ は型で，‘花子’，“花子”，‘30’，‘チェス’，‘テニス’，‘CS_2’，“情報工学科” がオブジェクトである．ψ-項同様にオブジェクト間には束構造が導入されており，継承が可能となっている．

また，3.4.1項の例のように，

E：アーク[始点 → X：ノード, 終点 → Y：ノード]

の事実から構成されるグラフを考えよう．O-論理では次のようなルールの記述

$add(E, nil)$：パス[始点 → X, 終点 → Y] ⇐
　　　　E：アーク[始点 → X：ノード, 終点 → Y：ノード].

$add(E, P)$：パス[始点 → X, 終点 → Y] ⇐

$$E：アーク[始点 \rightarrow X：ノード, 終点 \rightarrow Z：ノード],$$
$$P：パス[始点 \rightarrow Z：ノード, 終点 \rightarrow Y：ノード].$$

が可能である．特徴は，ルールによって新しく生成されるオブジェクト（識別子）$add(E,P)$ が，依存関係を明確にするために，項として記述されていることである．このようなデータベースに対する問合せ処理として，導出に基づく健全で完全な証明手続きが与えられている．つまり，演繹データベースの枠組みを保存した形での拡張となっている．

F-論理[23] はO-論理をさらに拡張したもので，型とオブジェクトの区別をなくし記述の自由度を増し，ラベルもオブジェクトとして扱うことによって，ラベル（スキーマ）に対する問合せやメソッドの記述を可能にしている．次の例を考えよう：

X[子供(Y) → {Z}] ⇐
　Y：人[子供_obj → 子供(Y)[メンバ → {Z：人}]],
　X：人[子供_obj → 子供(X)[メンバ → {Z：人}]].

⇐ の左の‘子供(Y)’，右の‘子供_obj’と‘メンバ’はラベルであり，オブジェクトとして記述されているのがわかる．⇐ の右の‘子供（Y）’は，⇐ の左のものとは役割が異なっており，役割によって異なった解釈をすることによって，（高階の意味論を避け）効率的な処理を考慮している．

O-論理や F-論理では，汎化関係は束構造として事前に与えられたものと位置づけられているが，DOT[7] では，仮想オブジェクトの記述と，束構造の動的の生成および汎化関係自体の推論を可能にしている．たとえば次の例を考えよう：

人[所属 → 組織]
学生：人[所属 → 大学]

仮想オブジェクトとは，学生の所属がわからないときでも，それを‘学生.所属’と表現できることである．汎化関係の推論としては，上の二つから，DOTでは‘大学：組織’，つまり‘大学 is_a 組織’の推論を基本的なメカニズムとしている．

Juan[29]（現在は Quixote）では，レコード・プログラミングの考えをさらに押し進め，F-論理や DOT の拡張と統合を試みている．

以上見てきたように，演繹データベースの拡張は，レコード・プログラミングという視点から見ると，非正規関係や複合オブジェクトのような構造データの記述とその問合せ処理から，型（オブジェクト）階層と継承機能，オブジェクト識別性の扱い，メソッドの記述，などさまざまな機能を持った言語に拡張されている．これら言語（論理）の間の関係についての研究も行なわれてきており，共通する理論の研究が今後の大きな課題となっている．

3.5 まとめ

レコード・プログラミングという，まだ明確ではないが，大きな可能性を持った“パラダイム”を，ψ-項とデータベースでの応用について，例を中心にして，駆け足で見てきた．3.1節で述べたように，レコード・プログラミングの応用はデータベース分野だけではない．自然言語の分野でも重要な地位を占めている．たとえば，本書では紹介できなかったが，ICOT で研究開発している談話理解システム DUALS の実装言語は，状況理論/状況意味論の考えを反映したレコード・プログラミング言語 CIL であり，その理論的基礎をレコード代数に置いている．

前節の最後で，レコード・プログラミングの下に横たわる共通の理論が大きな課題であると述べた．参考文献 19) は，いくつかの応用分野で使われているレコード・プログラミングについて，以下のような対応関係を示している：

C-論理[18)]＋発想推論＝ψ-項[12)]の推論
＝LOGIN[13)]，または
素性構造の推論[27)]

C-論理は複合オブジェクトのための言語であり，ψ-項はもともとは継承を中心にした知識表現を目指したものであり，素性構造は自然言語処理での素性(feature)の扱いを目的としている．このような対応関係は，多くの分野でのそれぞれ固有の意味を反映させた言語の共通のパラダイムとしてのレコード・プログラミングを浮かび上がらせつつある．

本章で概説したレコード・プログラミングのパラダイムは，実世界の複雑な構造を持ったオブジェクトを，計算機の上で，各問題領域の意味を反映したレ

コード構造として近似し，それを基にしたプログラミング言語を考える，と一口でいうことができる．このパラダイムは今後，プログラミング言語，知識表現言語，および データベース言語の三つの領域に またがった 新しい 言語として，より多くの応用分野で大きな役割を果たすだろう．

3.6 文献案内

レコード・プログラミングは，この数年の間に急速に研究が盛り上がっている分野であって，まだしばらくは大きな展開が予想されている．したがって，まだ教科書はないが，この分野の全体像をつかむには参考文献 10) あたりが適していよう．あとは個別の言語に関する 論文を読むのをお勧めする．ψ-項に関しては12)，13)，C-論理，O-論理，F-論理に関してはそれぞれ18)，22)，23)が読みやすい．大部であるが F-論理の集大成ともいうべきものが24)として出版されている．そのほかの言語として，DOT は7)， Juan は29)をお勧めする．

これらに関連した知識として，最近の傾向は含まれてはいないが，データベース・プログラミング言語のサーベイとして 14)，演繹データベース，オブジェクト指向データベースのそれぞれのサーベイとして 3)，6) が読みやすい．また関連の基礎知識としては，参考文献に掲載した井田哲雄編『新しいプログラミング・パラダイム』中のいくつかの文献が有用だろう．

参考文献

1) 井田哲雄編：『新しいプログラミング・パラダイム』，第4章　制約論理型プログラミング（相場　亮），pp. 65–89，共立出版，1989.

2) 井田哲雄編：『新しいプログラミング・パラダイム』，第8章　項書換えシステムと完備化手続き（大須賀昭彦），pp. 165–186，共立出版，1989.

3) 勝野裕文：演繹データベースの動向，情報処理学会アドバンスド・データベース・システム・シンポジウム，12月，1987.

4) 井田哲雄編：『新しいプログラミング・パラダイム』，第5章　オブジェクト指向（柴山悦哉），pp. 91–115，共立出版，1989.

5) 井田哲雄編：『新しいプログラミング・パラダイム』，第3章　マルセイユ Prolog，PrologII，PrologIII，（田中二郎），pp. 43–63，共立出版，1989.

6) 田中克己：オブジェクト指向データベース・システム――その概念と背景，**bit**，Vol. 20，No. 6，pp. 83–90，共立出版，1988.

7) 塚本昌彦，西尾章治郎，長谷川利治：オブジェクト指向の概念を導入した論理データベースの

ための項表現 DOT，情報処理学会アドバンスド・データベース・システム・シンポジウム，12月，1989.

8) 長尾　真，片山卓也，植村俊亮：『情報の構造とデータベース』，情報科学，8，岩波書店，1983.

9) 井田哲雄編：『新しいプログラミング・パラダイム』，第6章　型推論と ML（横田一正），共立出版，1989.

10) 横田一正，西尾章治郎：演繹・オブジェクト指向データベース，情報処理，Vol. 31，No. 2，pp. 234–243，1990.

11) S. Abiteboul and S. Grumbach : COL : A Logic-Based Language for Complex Objects, *EDBT ' 88*, in *LNCS*, 303, 1988.

12) H. Aït-Kaci : An Algebraic Semantics Approach to the Effective Resolution of Type Equations, *Theoretical Computer Science*, Vol. 45, pp. 293–351, 1986.

13) H. Aït-Kaci and R. Nasr : LOGIN : A Logic Programming Language with Built-in Inheritance, *Journal of Logic Programming*, Vol. 3, pp. 185–215, 1986.

14) M. P. Atkinson and O. P. Buneman : Types and Persistence in Database Programming Languages, *ACM Computer Surveys*, Vol. 19, No. 2, 1987.

15) F. Bancilhon and S. Khoshahian : A Calculus for Complex Objects, *PODS*, 1986.

16) C. Beeri, S. Naqvi, O. Shnueli and S. Tsur : Sets and Negation in a Logic Database Language (LDL), *PODS*, 1987.

17) P. Buneman and F. Bancilhon (ed.) : *Proc. of Workshop on Database Programming Languages, Tech. Report, MS-CIS-88-93*, Univ. of Pennsylvania, Nov., 1988.

18) W. Chen and D. S. Warren : C-logic for Complex Objects, *PODS*, 1989.

19) W. Chen and D. S. Warren : Abductive Reasoning with Structured Data, *NACLP*, 1989.

20) R. Hull, R. Morrison and D. Stemple (eds.) : *Proc. of Second International Workshop on Database Programming Languages*, Greneden Beach, Oregon, June 4–8, 1989.

21) B. E. Jacobs : On Database Logic, *JACM*, Vol. 29, No. 2, 1982.

22) M. Kifer and J. Wu : A Logic for Object-Oriented Logic Programming (Maier's O-logic Revisited), *PODS*, 1989.

23) M. Kifer and G. Lausen : F-Logic : A Higher-Order Languages for Reasoning about Objects, Inheritance, and Scheme, *SIGMOD*, 1989.

24) M. Kifer, G. Lausen and J. Wu : Logical Foundations of Object-Oriented and Frame-Based Languages, *SUNY TR 90/14 (revised)*, June, 1990.

25) W. Kim, J.-M. Nicolas and S. Nishio (ed.) : *Deductive and Object-Oriented Databases*, North-Holland, 1990.

26) D. Maier : A Logic for Objects, in *Preprint of the Workshop on Foundations of Deductive Databases and Logic Programming*, 1986.

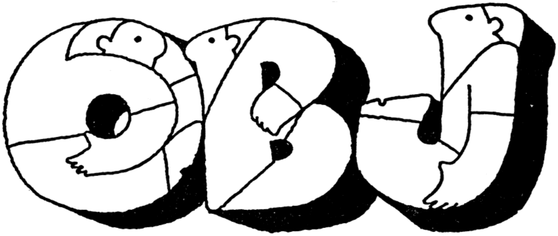
27) H. H. Porter, III : A Logic-Based Grammer Formalism Incorporating Feature-Structure and Inheritance, *Ph. D. Dissertation*, Oregon Gradiate Center, 1988.

28) K. Yokota : Deductive Approach for Nested Relations, in *Programming of Future Generation Computers II*, eds. by K. Fuchi and L. Kott, North-Holland, 1988.

29) K. Yokota : Outline of a Deductive and Object-Oriented Database Language Juan, 情報処理学会データベース・システム研究会，78-16，7月，1990.

（よこた かずまさ）

第4章　抽象データ型とOBJ2　　二木厚吉・中川　中



4.1 はじめに

抽象データ型の概念は1970年代に高まりをみせたソフトウェア開発法の改革運動の所産の一つである．この改革運動の中心的なテーマが構造化プログラミングであったことに象徴されるように，そこにおける最大の問題点は，高い信頼性と保守性を実現できるようなモジュール化を実現するためには，ソフトウェアの基本構造をどのように定式化すればよいかというものであった[1]．gotoなしプログラミング，データ＋アルゴリズム＝プログラム，データ型，抽象データ型といったキーワードの変遷が示すように，まずプログラムの制御のレベルからスタートした構造化は，データ構造を通じてのプログラムの構造化，データ構造とそれを扱う手続きを一体としたデータ型に基づく構造化，データ型を抽象化することでより汎用性を高めた抽象データ型に基づく構造化というように発展してきた．現在，ソフトウェアの開発現場において最も有望視されているプログラミング・パラダイムの一つであるオブジェクト指向の概念も，この抽象データ型の概念の延長線上にあるという見方ができる．抽象データ型とオブジェクト指向との関係については，すでに発行ずみの『新しいプログラミング・パラダイム』（共立出版）に柴山氏の解説があるのでそちらを参照されたい．

代数的仕様記述法 (algebraic specification technique) は，抽象データ型の仕様を厳密に記述する方法として1975年ごろに登場し，それ以後それに関連して理論

から応用まで広い範囲にわたり活発な研究が展開されてきた．現在では要求仕様からコーディングまでのソフトウェア開発のさまざまな局面において代数的仕様記述法に基づいた活動が盛んである．本章では，代数的仕様記述法に基づいて設計開発された代数型言語 OBJ2 について解説する．関数型言語や論理型言語という名前が，アルゴリズムの記述のレベルにおける関数や論理という計算機構のモデルに由来するのに対し，代数型言語の名前はソフトウェアのモジュール化のレベルにおける代数というソフトウェア・モジュールのモデルに由来しているのに注意されたい．一方，代数型言語の計算機構のモデルは，代数系を構成する関数の集合に対し操作的な意味を与える項書換えシステムである．項書換えシステムは，関数に対しリダクションに基づく操作的な意味を与える一般的な方法を提供する．この意味で，代数型言語は関数型言語の一種であるということもできる．

以下では，まず OBJ2 の基盤にある抽象データ型と代数型言語の考え方について簡単に説明し，次に OBJ2 の基本構造とその特徴，特に一般の代数型言語に対する拡張である，オブジェクトのパラメータ化と順序ソートなどについて，例を中心に解説する．

4.2　抽象データ型と代数型言語

ここでは，代数型言語が提唱される契機となった抽象データ型の考え方を簡単に説明し，代数型言語の基本事項について解説する．

4.2.1　抽象データ型

抽象データ型の考え方は，ある特定の種類のデータを操作する関数を一箇所に集め，そのデータの表現形態を外部から遮断し，そのデータを操作する関数の呼出し形態のみを知らせる，というように要約できる[1]．この方法は一般に**データ隠蔽**（data encapsulation）と呼ばれ，関数定義を含めたデータと関数を一体とした実体を，通常**データ型**または**抽象データ型**と呼ぶ．データ型が具体的なデータと関数を一体化した具体的な対象であるのに対し，抽象データ型は抽象化したデータと関数からなる抽象化された対象であるという差がある．この

考え方の利点は，データの表現（実現）形態の影響が局所化されるため，プログラムの設計，変更，そのほかの操作が容易になることである．抽象データ型を構成する，幾種類かのデータの集合とその上の関数群は代数（系）として自然にモデル化できる．簡単な例で見てみよう．

スタックを抽象データ型で実現する場合を例にとれば，通常のプログラミング言語では，

●スタックの表現形態

●スタックへのアクセス形態

を特定する必要がある．表現形態は，その言語の特性や処理効率を勘案して，配列と添字（ポインタ），リストなど，基本データタイプないしその組合せから選ばれる．アクセス形態は，定められたスタックの表現形態を前提として，関数や手続きの呼出しパターン（型）を示すことで与えられる．たとえば，

push（E：要素，S：スタック）：スタック

top（S：スタック）：要素

pop（S：スタック）：スタック

のように表わされる．それぞれの関数の意味は自然語でコメントされるであろう．すなわち，表現形態への依存を払拭しきれない点，操作の正確な意味は結局コードに現われる処理手順からしか得ることができない点において，不完全なデータ抽象にならざるをえない．

これに対して，代数によるスタックのモデルはデータ抽象の実現そのものである．このモデルでは，表現形態に依存しないかたちで，スタックへの操作は，

push は，ある要素とあるスタックから，別のあるスタックを作る，

top は，あるスタックから，ある要素を取り出す，

pop は，あるスタックから，別のあるスタックを作る，

の三つにモデル化され，それらの意味は，

top (push (ある要素，あるスタック))＝その要素 (a)

pop (push (ある要素，あるスタック))＝そのスタック (b)

のように二つの等式で表現される．等式(a)は「あるスタックにある要素を push して得られるスタックを top して取り出される要素は，push されたその要素に

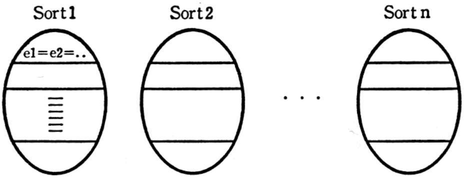


図 4.1　多ソート代数モデル

ほかならない」と述べている．等式(b)も同様である．このように代数モデルは，関数のタイプ（引数と返却値のデータタイプの系列）および関数を合成して示すことができる値の間に成立する合同関係（congruence relation）だけを定義することで，対象を規定する．つまり，データの表現形態を特定しないまま，それへの操作の意味を宣言的かつ正確に記述できるのである[2),3)]．

4.2.2　代数型言語

代数型の言語では，まず記述したい世界を構成する対象（またはデータ）をその種類（sort）により分類し幾つかの集合に分割する．その上で同一種類の対象の間に同等関係を定義することで，記述したい世界の意味を与える(**図4.1**)．同一種類の対象の集合を**ソート**（sort）と呼ぶ．ソートは本来対象の種類を区別するための名前であるが，以下ではある同一種類の対象の集合そのものもソートと呼ぶことにする．各ソートに属する任意の対象（データ）はすべて定対象（constant）に対し適当な（一般には複数個の）関数を何回か適用して得られる．たとえば，自然数というソートNatに属する任意の対象（自然数）は，1を加えた次の自然数を値とする後者関数をsucと表わすことにすると，

suc(suc(…suc(0))…)

のようにsucを定数0に対して何回か適用することで得られる．代数型言語の基本をなすモデル化のこうした枠組みは，（多ソート）**代数**と呼ばれるものであり，ソフトウェア・モジュールの汎用モデルと考えられている．抽象データ型はまさにこの代数である，というのが代数的仕様記述法の基本をなすテーゼである．

まず簡単な例として，自然数とブール値，およびそれらの上のいくつかの関

数からなる世界を，代数として以下のようにモデル化してみよう．まず，二つの集合，Nat と Bool を与える．

対象ソートの集合＝{Nat, Bool}

次に，各ソートの定数を与える．

Bool＝{true, false}

Nat＝{0}

次に，必要な関数の型を宣言する．関数の型とは，一般には複数（0の場合もある）の引数の並びと返却値のソートの対を指し，一般には

引数ソート1　引数ソート2…引数ソートN->返却値ソート

のように表わす．まず，1を加えた次の自然数を値とする関数 suc を，次に，加算，乗算，大小比較，および否定を考える．また，定数も引数を持たない関数と考えれば，上記の定数も含めて，各関数の型は以下のように記述できる．

```
true  : -> Bool
false : -> Bool
0     : -> Nat
suc   : Nat ->Nat          …後者関数
+     : Nat Nat -> Nat     …加算
*     : Nat Nat -> Nat     …乗算
<     : Nat Nat -> Bool    …大小比較
~     : Bool -> Bool       …否定
```

0引数関数としての定数も含めた上のような関数の型宣言の集合を**シグネチャ** (signature) という．最後に，これらの（定数を含めた）関数の意味，つまり合同関係を等式で次のように与える．ここで，関数記号＋，＊，＜は中置記法，~は前置記法で用いている．また，NとMはNat型の変数である．

$$N+0=N$$
$$N+\mathrm{suc}(M)=\mathrm{suc}(N+M)$$

$$N*0=0$$
$$N*\mathrm{suc}(M)=(N*M)+N$$

$$0<0=\mathrm{false}$$

0 < suc(N) = true
suc(N) < 0 = false
suc(N) < suc(M) = N<M

~true=false
~false=true

等式集合で与えられる各関数の意味は，始代数という概念を用いて純粋に宣言的，あるいはモデル論的に定義できる．それはまた関数記号から構成される項（term）の集合上の合同関係として解釈できる．また一方では，等式の集合は項書換えシステムにおける書換え規則として操作的に解釈できる．以下で，このような代数型言語の宣言的な意味と操作的な意味について概説する．

4.2.3　始代数

前項のようにして記述したい世界を，その世界を構成する幾つかのソートとその上の合同関係として「代数的に」モデル化したとしよう．このようなモデル化の結果得られる「モデルの世界」（または代数の世界）と，ソート（名），型付きの関数記号，等式の三項目で構成されるその「記述の世界」（または仕様の世界）との間には，次のような関係が暗黙裡に仮定されていると考えられる．

（1）　**非冗長性**　モデルの世界のどのような対象も，記述の世界において，関数記号（定数を含む）を用いて関数合成により構成できるある項に対応している．つまり，記述の世界で表現できないような対象はモデルの世界に存在しない．

（2）　**非曖昧性**　モデルの世界である二つの対象が等しいときそしてそのときに限り，記述の世界での等式に基づいてその二つの対象が等しいことを示すことができる．

（1）は冗長で不必要な対象（データ）が存在しないことを保証し，（2）は二つの対象（データ）が偶然に等しくなるような紛らわしい事態が生じないことを保証する．これらの性質は，記述自体とそれが記述しているもの，つまりその記述のモデル，が整合または一致しているための最低限の条件であり，特に

仕様記述の場合に重要である．簡単な例で見てみよう．

［**非冗長性**］ 文字コード体系を決める場合，大文字子文字間の変換を単純なビット操作で行なえるなどの理由で，コード間に不必要な間隔を置くことがある．そうして生じるどの文字にも対応しないコードも，実現レベルでは文字タイプのデータに（一見）含まれるであろうが，読み書きなどで使用されない限り，特に問題はない．しかしながら，仕様の段階ではあるコードが文字データに含まれるか否かを明確にしておく．つまり，文字コードのセットに含まれる要素に冗長性を持たせず，限定的に明示することが重要である．たとえば，ある実現が表わしている関数のふるまいが，仕様に記述されている対応する関数のふるまいに反していないかを調べることは，仕様に示されているコード以外の文字コードは存在しないという前提なしでは意味がない．

［**非曖昧性**］ 数え上げタイプのデータ，たとえば性別や所属部門，をプログラムで扱う場合，女性を1，男性を2，そのほかを3とする，などのように便宜的に自然数の割当てで代用することが多い．その際，異なるものに同じ数を割り当てても，たとえば女性と経理部とに等しく1を割り当てても，両者が混同されない限り，すなわち，「女性と経理部が等しければ」というような扱いがなされない限り，実現レベルでは問題はない．しかしながら，仕様の段階では偶然に等しいデータが存在するような曖昧性は存在しないという前提が重要である．なぜならば，仕様の中で成立する関係が実現の中でも成立しているかを確かめる行為は，こうした偶然の関係が存在すれば，無に帰するからである．

代数型言語は，この二つの性質を備えたモデルの世界，つまりその言語で記述された代数，が一意に定まるという際だった特徴を持つ．この点をもう少し細かく見てみよう．前項のような記述が与えられたとき，定数や関数記号の解釈をいろいろに変えることで，この記述を満たす多数の代数を構成することができる．それらの代数の中で，

（1） どのソートに対しても，記述で用いた関数記号を用いて関数合成により構成できないような要素は存在しない，

（2） 要素間の合同関係は記述された等式集合から導くことのできるものだけであり，それ以外の要素間の合同関係は存在しない，

の二つの性質を持つ代数はとくに **始代数** と呼ばれる†．この二つの性質は先の非冗長性と非曖昧性の言いかえにほかならない．代数型の言語の大きな特徴は，その言語で書かれたどんな記述に対しても，それを満たす代数の中に始代数がただ一つ存在することである[2),3)]．

4.2.4　項代数

記号的に項を構成する操作そのものを関数の解釈に用いることで **項代数** (term algebra) と呼ばれる特別な代数を次のように構成できる．ここでは項を表現するのに標準的な f(t1, t2, …, tn) のような記法を用いることにする．

（1）　定数はそれを表わす記号そのものと解釈する．

（2）　関数 f の引数 t1, t2, …, tn に対する値は f(t1, t2, …, t*n*) と解釈する．

たとえばシグネチャ

　a ：-＞S　… 定数を表わしている

　f ：S-＞S

から構成される項代数のソートSの項は，以下のようになる．

　a , f(a), f(f(a)), f(f(f(a))), …

ここで項と呼んでいるものは通常，**定数項** (constant term) または **閉項** (closed term) と呼ばれるものである．関数の意味を定義する等式は，項代数を構成する項の間に合同関係を定義する．この際，先に述べたように，「与えられた等式から導くことのできない要素間の合同関係は存在しない」ようにする．その結果等しいとされる要素を同一視し，もとの代数の商代数を作ることで先に述べた始代数と同型な項代数が得られる[2),3)]．この商代数が代数型言語の宣言的な意味，つまりその言語のモデル論的な意味として広く受け入れられるに至っている．代数型言語は抽象データ型言語を記述するための言語であるという立場に立てば，この商代数が抽象データ型の宣言的な意味を与えることになる．

たとえば，上の例において等式の集合が，

　f(f(X))＝X　… Xは変数

だけであるとすると，項代数の要素つまり定数項の集合は，項aに等しい定数

---

† これは始代数の定義ではない．しかし，この性質が始代数の特徴づけとして必要十分であることが示されている．

項をすべて集めた集合（これを[a]と記することにする），

$$[a]=\{a, f(f(a)), f(f(f(f(a)))), \cdots\}$$

と項 f(a) に等しい定数項をすべて集めた集合（これを[f(a)]と記すことにする），

$$[f(a)]=\{f(a), f(f(f(a))), f(f(f(f(f(a))))), \cdots\}$$

のような二つの集合に分割される．したがって，この等式が定義する商代数は[a]と[f(a)]に対応する二つの要素を持ち，関数 f は，

$$f([a])=[f(a)]$$

$$f([f(a)])=[a]$$

のように定義される．

4.2.5 項書換えシステム

OBJ2 のような代数型言語に対しては，その実行系の基礎になる操作的モデルとして，項書換えシステムが用いられる．ここでは説明を割愛するが，代数言語で用いる等式がある条件を満たせば，代数型言語に対して与えられる始代数に基づく宣言的意味と，項書換えシステムに基づく操作的意味が一致することが知られている．

項書換えシステムでは，与えられた等式は書換え規則とみなされ，左辺から右辺へ，と方向性を持つことになる．また，項書換えシステムでは有限個の書換え規則ですべての計算が定められるが，計算能力はどんな計算可能な関数も計算できるだけの一般性があり[4]，また自動推論システムとしても使える．さらに，この実行方式の「呼出し」は，一方を基底項に制限するパターンマッチングによって行なわれるため，単一化のように処理が重たくなりやすいという欠点もない．項書換えシステムの詳細については，『新しいプログラミング・パラダイム』（共立出版）の大須賀氏の解説などを参照されたい．

4.3 OBJ2

OBJ2 は 4.2 節で述べたような代数型の言語に属し，代数型言語一般が有する以下のような利点を持つ．

（1）　データ抽象を自然に表現でき，それを通じ問題を任意の抽象化レベルで記述できる．

（2）　始代数で与えられる宣言的な意味と，項書換えシステムで与えられる操作的な意味をともに厳密に与えることができる．

また，OBJ2は最近のプログラミング手法を多く取り入れ，パラメータ化オブジェクト，対話型プログラミングなどを実現しており，以下のような特色を備えた記述能力の高い言語となっている．

（1）　関数記法の混在

（2）　高度のパラメータ化

（3）　順序ソート

（4）　交換則，結合則を加味してのパターンマッチング

（5）　引数評価の順序の指定

（6）　複雑なモジュール表現

以下では，まずOBJ2の基本構造を説明し，これらの特色を順次例解していく．

4.3.1　OBJ2の基本構造

OBJ2のプログラムはモジュールを単位とする階層構造に組み立てられる[5),6)]．モジュールにはobjectとtheoryとがあり，theoryは4.3.4項で述べるパラメータ化で用いられる．ここでは，簡単なobject（**図4.2**）とその階層構造の例を**図4.3**に示す．

代数型言語の一般の説明ですでに述べたとおり，OBJ2の一つのモジュールは以下の記述を含む．

（1）　ソート宣言（sort … の行[†]）

（2）　関数（定数を含む）の型宣言（op … の行）

（3）　等式による関数の意味の定義（eq … の行）

たとえば，モジュールSTATESでは，ソートStateとその定数belgiumなどが宣言され，espanaとspainが等号で結ばれている．等式に与えられているState

† ここで「行」としているが，厳密には（原則として）終端符．ピリオドで終わる語の列がOBJ2の構成要素であり，「文」とすべきかもしれない．

```
object STATES is
  sort State .
  op (the united kingdom) : -> State .
  op belgium              : -> State .
  op france               : -> State .
  op spain                : -> State .
  op espana               : -> State .
  eq State : espana = spain .
endobj

object DENSITY is
  protecting STATES .
  protecting INT .
  op population : State -> Int .
  op area       : State -> Int .
  op density    : State -> Int .
  var S : State .
  eq Int : population(the united kingdom) = 56280000 .
  eq Int : population(belgium)            =  9860000 .
  eq Int : population(france)             = 54220000 .
  eq Int : population(spain)              = 37930000 .
  eq Int : area(the united kingdom)       =   244100 .
  eq Int : area(belgium)                  =    30513 .
  eq Int : area(france)                   =   547026 .
  eq Int : area(spain)                    =   504782 .
  eq Int : density(S) = population(S) quo area(S) .
endobj
```

図 4.2

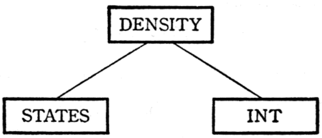


図 4.3　モジュールの階層構造の例

という宣言は，等式が扱うソートを示しており，文脈から明らかなときは省略してもよい．

モジュールはほかのモジュールを参照することができ，参照関係を通じ，モジュールの階層構造†ができる．モジュール DENSITY における protecting… の行はほかのモジュール，ここでは STATES と INT を参照した例である．このモジュールは新たなソートを宣言せず，STATES のソート State 上の関数 population, area, density を宣言し定義している．モジュール INT は OBJ2 の組込みモジュールで，整数とその四則関数，大小比較を取り扱う．density を定義す

† 循環的な参照は，一般に意味づけが困難であるので，認めていない．

る等式においてState型の変数Sが用いられているが，これは「ソートStateのすべての要素について，…」を意味する．一般に，等式に現われる変数はすべて全称的であり，「すべての…について」を意味する．

ほかのモジュールを参照する場合を加味すると，どんな宣言や定義があるモジュールに属するかを定める規則は以下のようにまとめることができる．

ソートの宣言　あるモジュールで直接（sort 行により）宣言されたソートはそのモジュールに属する．そのモジュールが参照しているモジュールに属するソートも同様である．

関数の宣言　あるモジュールに属するソートについて，直接（op 行により）宣言した定数および関数は，そのモジュールに属する．そのモジュールが参照しているモジュールに属する定数および関数も同様である．

等式による関数の定義　そのモジュールで直接（eq 行により）定義された等式は，そのモジュールでの合同関係を規定する．そのモジュールが参照しているモジュールで定義された等式も同様である．

4.3.2　モジュールの参照方法

モジュールの参照方法には次の3通りがある．

（1）　**protecting**　参照されるモジュールに（直接またはそれ自身の参照により）属するすべてのソートにつき，要素を新たに付け加えたり，合同関係を変化させたりしない．最も制約のきつい参照方法である．

（2）　**extending**　参照されるモジュールに属するすべてのソートにつき，合同関係は保存するが，要素を追加する可能性がある．すなわち，あるソートを「拡大」することはある．

（3）　**using**　（1）や（2）のような制限なしに，参照されるモジュールを自由に使う．

モジュールには，それに属するソート上に成立する合同関係を定める（商）代数が対応する．たとえば，さきのモジュールSTATESには，

ソートの集合＝{State}

State＝{the united kingdom, belgium, france, spain(＝espana)}

という代数が対応する．protecting により このモジュールを参照する場合は，

ソート State はこの四つの相異なる要素からなる集合のまま変化しない. extending による参照では, ソート State がこの四つの相異なる要素を持つ点は変わらないが, 新たな要素が付加されることがあり得る. たとえば,

```
object EXTENDEDSTATES is
  extending STATES .
  op norway  : -> State .
  op andorra : -> State .
endobj
```

という参照をされた場合, ソート State は

State＝{the united kingdom, belgium, france, spain(＝espana), norway, andorra}

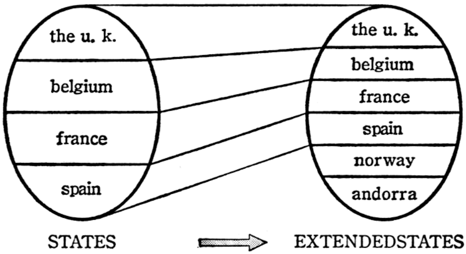


図 4.4 extending の例

という六つの要素からなる集合に変化する (図 4.4). using による参照では, もとの四つの要素の存在すら保証されない. たとえば,

```
object USEDSTATES is
  using STATES .
  op norway : -> State .
  op andorra : -> State .
  eq : belgium = france .
endobj
```

という参照をされた場合, ソート State は

State＝{the united kingdom, france(＝belgium), spain(＝espana),
norway, andorra}

というように, もとの 2 要素が一つに縮退し, 新たな要素を加え五つの要素からなる集合に変化する (図 4.5). こうした変化をソート State 上に宣言された関数から見ると, 定義域がどう変化したかに応じて,

●protecting では, 関数の定義を変更する必要はない.

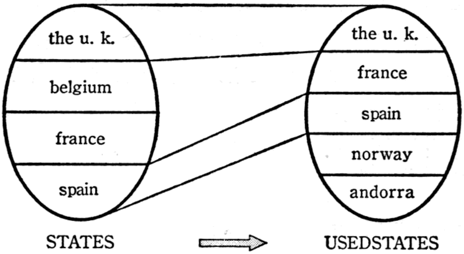


図 4.5　using の例

- extending では，従来の定義はそのままでよいが，新たな要素を取り扱うための等式が必要になることがある．
- using では，従来の定義自体に手を加える必要も生じ得る．

となる．また，ソートの同一性という観点では，protecting で参照しているモジュール間では参照されたモジュールから受け継いだソートは同一であるのに対し，extending で参照しているモジュールが受け継ぐソートはもとのソートあるいは protecting で受け継がれるソートを包含する．using では，もとのソートとの関連は失われる．

protecting や extending による参照におけるこうした制約条件の成立を証明することは容易ではない．たとえば，新たな要素が加えられたかどうかの判定は，単に op 行によって新たな定数や関数が宣言されたかどうかの判定とは異なる．さきの STATES を参照する例を続けると，

```
object STATES' is
  protecting STATES .
  op (kingdom of spain) : -> State .
  eq State : (kingdom of spain) = spain .
endobj
```

では，ソート State に新たな定数を宣言しているが，等式により既存の定数に吸収され，換言すれば，等式により拡張された合同関係において既存の定数と合同になり，結果としては State は変化しない．すなわち，あるソートに新たな要素が加えられていないことは，以下の手順に従って判定される．

（1）　op 行により定義された，そのソートに属する定数のおのおのについて，eq 行の等式により拡張された合同関係において既存の要素（項代数

における項）の少なくとも一つと合同になること．

（2） op 行により定義された，そのソートを返却ソートに持つ関数のおのおのについて，その関数記号から構成し得るすべての項が，eq 行の等式により拡張された合同関係において既存の項の少なくとも一つと合同になること．

一般には，（1），（2）を自動的に証明することは困難であり，適切な帰納条件を選び帰納法により証明することになる．合同関係の保存の判定の自動化はさらに困難である．

4.3.3 混置関数記号

OBJ2 では，関数の記法を自由に定義でき，プログラムの文章としての読みやすさを高めることができる．図 4.6 に例を示す．

図 4.6 のモジュールでの関数の宣言では，

and (a)

not_ (b)

if_then_else_fi (c)

のように，記法が指定されている．下線符は引数の位置を示すので，これはこれらの関数の記法を次のように定めていることになる．

（a） true and false, A and B, …

```
object BOOL is
  sort Bool .
  sort ASort .
  op true : -> Bool .
  op false : -> Bool .
  op _and_ : Bool Bool -> Bool [1] .
  op _or_ : Bool Bool -> Bool .
  op not_ : Bool -> Bool [2] .
  op if_then_else_fi : Bool ASort ASort -> ASort .
  vars X Y : ASort .
  vars A B : Bool .
  eq ASort : if true then X else Y fi = X .
  eq ASort : if false then X else Y fi = Y .
  eq Bool : not A = (A == false) .
  eq Bool : A and B = if A == true then B else false fi .
  eq Bool : A or B = if A == false then B else true fi .
endobj
```

図 4.6

（b）　not false, not A, …

（c）　if true then X else Y fi, …

通常の用語を用いれば，and は中置記法，not は前置記法である．形式的には，関数記法の定義は，

［予約語1］［引数1］［予約語2］［引数2］…

のようになる．ただし，中括弧は省略可能を表わし，また，少なくとも一つの予約語または引数は必要である．すなわち，

op _ : ASort -> AnotherSort.

op a : -> ASort.

のような宣言は認められるが，

op : -> Asort.

のような宣言は認められない（そもそも，意味を与えようがない）．さらに，引数ソートの並びの数は下線符の数と一致しなければならない．こうして自由に定義される記法に従って，任意の文字列を構文解析することは，動的に変化する文脈自由文法を取り扱うことにほかならない．

特に記法を指定しなければ，通常の関数記法，たとえば f(a, g(b)) など，が用いられる．

関数 and や not の宣言に見られる [1] のような記述は，関数の優先順位を表わす．特に指定のない関数は優先順位 0 であり，数字の大きいほど優先順位が高いから，この例では not > and > or であり，

A and not B or not C and D

は

(A and (not B)) or ((not C) and D)

と同じ項を表わす．

4.3.4　モジュールのパラメータ化

OBJ2 には object をパラメータ化し，その使用価値を高めたり再利用を促進したりする機能がある[5),7)]．強力なパラメータ化を実現するために，仮パラメータに埋め込まれる実パラメータに，構文上ばかりでなく意味上の制約を記述するために theory という記述単位が用意されている．また，実パラメータの組

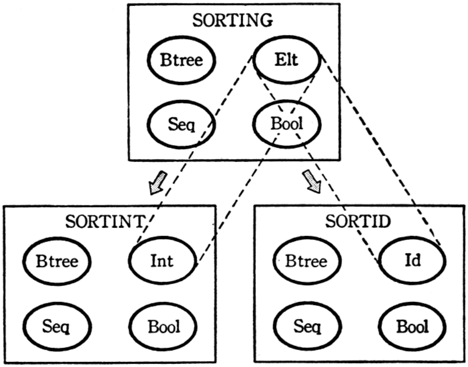


図 4.7 パラメータの使用例

込みの仕方に柔軟性を持たせるために，実パラメータの仮パラメータへの束縛の仕方を記述する view という記述単位も用意されている．まず，パラメータ化を使用した例を示す（図 4.7, 4.8）．

図 4.8 は，全順序が定義された任意のデータの集合に対し，そのデータの列 (Sequence) を全順序に従って整列された列を作り出す，パラメータ化されたモジュール SORTING を示している．

SORTING が参照しているモジュール SEQ もパラメータ化されたモジュールであり，どんな種類のデータに対しても有効な列の定義を与えている．列のパラメータに対する制約を表わしている theory TRIV は，単に「Elt に対応するソートが少なくとも一つあること」を要求している．

モジュール TOSET も，特定の対象を記述する object ではなく，特定の制約を満たす object の集合を表わす theory の一例である．TOSET の課す制約は，(Elt と仮称された) あるソートが存在し，そのソート上に（仮に＜と記された）全順序関係があることである．theory に具体的に記述すべき内容は object のそれとほぼ同じである．ただし，それ自体は実行対象ではないので，項書換えシステムとして望ましい性質などは考慮する必要がなく，等式の記述などに（事実上の）自由度が大きい．

二分木を用いた分類を行なうモジュール SORTING は，この theory TOSET

```
theory TRIV is sort Elt . endth

obj SEQ[X :: TRIV] is
  protecting BOOL .
  sorts Seq .
  op . : -> Seq .
  op __ : Elt Seq -> Seq .
  op _$_ : Seq Seq -> Seq .
  vars S S' : Seq .
  var E : Elt .
  eq : (. $ S) = S .
  eq : (E S) $ S' = E (S $ S') .
endo

theory TOSET is
  protecting BOOL .
  sort Elt .
  op _<_ : Elt Elt -> Bool .
  vars E E' E'' : Elt .
  eq : E < E = false .
  eq : (E < E') or (E' < E) or (E == E') = true .
  eq : not(E < E' and E' < E'')or(E < E'') = true .
endth

object SORTING [ELT :: TOSET] is
  extending SEQ[ELT] .
  op sort_ : Seq -> Seq .
  sort Btree .
  op makeBtree : Seq -> Btree .
  op flatten : Btree -> Seq .
  var L : Seq .
  eq : sort(L) = flatten(makeBtree(L)) .
  op nil : -> Btree .
  op _#_#_ : Btree Elt Btree -> Btree .
  vars E E' : Elt .
  eq : makeBtree(.) = nil .
  op insert_to_ : Elt Btree -> Btree .
  eq : makeBtree(E L)
       = insert E to makeBtree(L) .
  vars T T' : Btree .
  eq : insert E to nil = nil # E # nil .
  eq : insert E to (T # E' # T') =
       if E < E' or E == E'
       then (insert E to T) # E' # T'
       else T # E' # (insert E to T') fi .
  eq : flatten(nil) = (.) .
  eq : flatten(T # E # T')
         = flatten(T) $ (E flatten(T')) .
endobj

object SORTINT is
  protecting SORTING[INT] .
endobj

object SORTID is
  protecting SORTING[ID] .
endobj
```

図 4.8

で規定される抽象的な分類対象を仮定して，分類という仕事を記述している．整数や名前を分類する SORTINT や SORTID という実際の分類モジュールは，theory TOSET で示される仮パラメータに INT や ID といった実際の対象を埋め込むことで定義されている．別の言い方をすると，TOSET が課す制約を満たす任意の object の分類方法がパラメータ化された SORTING に記述され，その制約を満たす整数モジュール INT，名前モジュール ID をそれぞれ実パラメータとして束縛することにより，SORTINT や SORTID では，整数 INT や名前 ID を分類する object が定義されている．モジュール ID は名前 (identifier) とその上の順序関係を扱う組込みモジュールである．

この例に見られるように，OBJ2 におけるパラメータ化のスキームは以下のようにまとめることができる．

（1） theory の定義

（2） パラメータ化 object の定義

（3） 実パラメータ用 object の定義

（4） （3）の（2）への束縛による，実 object の定義

theory が課す制約は，形式的には，以下のようになる．

（1） ソートの存在

（2） （1）を介して規定される関数や定数の存在

（3） （2）から得られる項の間の合同関係

したがって，ある object がある theory の制約を満たすかどうかの判定は，以下の条件の判定に還元できる．

（1） その theory が仮定しているソートに対応するソートがその object に少なくとも一つ存在すること†.

（2） その theory が仮定しているあるソートの定数がその object の（1）によって対応づけられたソートに少なくとも一つ存在すること．また，その theory が仮定している関数が，その object の（1）によって対応づけられたソート上に少なくとも一つ存在すること．

（3） その theory が仮定している合同関係が，その object の（2）によって対

† 後に見るように，OBJ2 のソートには包含関係があるので，正確にはソート間の包含関係の成立も制約条件に含まれる．

応づけられた定数や関数から得られる項の間で成立すること．

このうち，(1)，(2)をviewと呼ばれるメカニズムを用いて使用者が指定することができる．例を示す．

```
view INTDESC of INT as TOSET is
  sort Elt to Int .
  vars X Y : Elt .
  op : X < Y to : Y < X .
endview

object SORTINTDESC is
  protecting SORTING[INTDESC] .
endobj
```

INTDESC は theory TOSET と object INT のソートと演算を対応づけている．すなわち，TOSET におけるソート Elt を INT におけるソート Int に，また，TOSET における関数

op _<_ : Elt Elt -> Bool.

を INT における関数

op _>_ : Int Int -> Bool.

に対応させている．object SORTINTDESC はこのマッピングを介して INT を SORTING のパラメータに束縛し，降順の整数分類モジュールを定義したものである．

明示的に view が与えられない場合，ソートと関数の自然な対応から定まる「暗黙の view (default view)」が用いられる．さきの SORTINT, SORTID は TOSET からそれぞれ INT, ID への暗黙の view を用いている．

(3)の合同関係の成立を自動的に判定することは一般に困難であり，参照関係の判定で述べたような人間の適当な介在が必要になる．

4.3.5　パラメータ化機構による高階関数の記述

OBJ2 の theory を用いたパラメータ化機能は，単に実パラメータの束縛に制約を与える目的に有効であるにとどまらない，たとえば，theory における関数の存在の仮定を関数の型宣言と捉えれば，高階関数の実現などに利用することが可能である[8]．例を示す．

```
object LIST[X :: TRIV] is
  sort List .
  op nil : -> List .
  op __ : Elt List -> List .
endobj

theory FN is
  sort S .
  op f : S -> S .
endth

object MAP[F :: FN] is
  protecting LIST[F] .
  op map : List -> List .
  var X : S .
  var L : List .
  eq : map(nil) = nil .
  eq : map(X L) = f(X) map(L) .
endobj
```

theory FN は「あるソートSの上にそのソートを引数と返却値とに持つ一項関数 f が定義されたデータ型（つまり代数）」を要求する．object MAP はこうしたデータ型Fの存在を仮定して，Fから構成されるリスト構造を LIST[F] として作り，これを用いてSの要素のリストからその要素おのおのに関数 f を適用したリストを求める関数 map を定義している．リスト構造は object LIST により定義され，object MAP はこれを利用している．

MAP の使用法の説明のために，整数上に sq（平方），x2（2倍）を計算する関数が定義された次の object FNS を考える．

```
object FNS is
  protecting INT .
  ops (sq_) (x2_) : Int -> Int .
  var N : Int .
  eq : sq N = N * N .
  eq : x2 N = N + N .
endobj
```

この FNS を theory FN に束縛する方法は二つあり，それはそれぞれ次のような二つの view V1, V2 で示すことができる．

```
view V1 of FNS as FN is
  sort S to Int .
  var X : S .
  op S : f(X) to : sq X .
endvw

view V2 of FNS as FN is
  sort S to Int .
  var X : S .
  op S : f(X) to : x2 X .
endvw
```

この二つのviewを用いてFNSをMAPの仮パラメータFに束縛することで，整数のリストの要素おのおのを平方または2倍した新しいリストを作る関数を提供するobject MAPSQまたはMAPX2を次のように定義することができる．

```
object MAPSQ is
  protecting MAP[V1] .
endobj

object MAPX2 is
  protecting MAP[V2]
endobj
```

この例からわかるように，MAPにおける関数mapは，任意の関数をしかるべきソートのリスト上に展開する，「関数 -> 関数」型の高階の関数を定義している．

パラメータ化のほかの応用としては，theoryとobjectとの関係を仕様とその実現と捉え，プログラム開発における段階的詳細化手法での詳細化の一ステップの検証方法を定式化することなどがある[9]．

4.3.6 順序ソート

OBJ2ではソートの間に包含関係を定義することができる．この意味で，OBJ2は多ソート等号論理ではなく順序ソート等号論理に基づいている[5]．順序ソートという言葉はソートの包含関係を半順序とみなせる事実に由来する．ソートに包含関係を持たせることにより，エラーや例外の処理，関数記号の重複，関数の多重継承などの記述が可能となり，多ソート論理への強力な拡張となっている．例としてSTACKINTを順序ソートを用いて記述した仕様を示す（図4.9）．スタックの定義は，要素をパラメータとして行なうべきであるが，ここでは説明を簡単にするためスタックの要素を整数に固定している．

```
object STACKINT is
  protecting INT .
  sorts Stack NeStack .
  subsorts NeStack < Stack .
  op empty : -> Stack .
  op push_to_ : Int Stack -> NeStack .
  op top_ : NeStack -> Int .
  op pop_ : NeStack -> Stack .
  var E : Int .
  var S : Stack .
  eq : top push E to S = E .
  eq : pop push E to S = S .
endobj
```

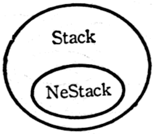


図 4.9 ソートの包含関係の例(1)

この例では，top や pop の定義域を空でないスタックに制限することでそれらの関数の意味を規定している．そのために導入されたソート NeStack は空でないスタックの集合であり，空スタックを含む Stack の下位のソートと宣言されている．この STACKINT では，空スタック上の top や pop はエラーというより，そもそも正しい項と見なされない（構文解析できない），として退けられる．すなわち，これらの関数をスタック上の部分関数としたわけである．

図 4.10 は順序ソートのメカニズムで関数の多重継承を記述した例である．

モジュール FACTORY（工場）で宣言されたソート Capital などは図 4.11 のような包含関係を持つ．あるソート上に定義された関数はその下位のソートに

```
object FACTORY is
  protecting INT .
  sorts Capital Depreciable Equipment Lathe .
      :
  subsorts Lathe < Depreciable Equipment .
  subsorts Depreciable < Capital .
      :
  op historiccost : Capital -> Int .
  op depreciationrate : Depreciable -> Int .
  op depreciationcost : Depreciable -> Int .
  op neededspace : Equipment -> Int .
      :
  ops alathe anotherlathe yetanotherlathe : -> Lathe .
      :
  var L : Lathe .
  var D : Depreciable .
  eq : historiccost(alathe) = 20000 .
  eq : historiccost(anotherlathe) = 25000 .
  eq : historiccost(yetanotherlathe) = 40000 .
  eq : depreciationrate(L) = 25 .
  eq : depreciationcost(D)
        = (depreciationrate(D) * historiccost(D)) quo 100 .
  eq : neededspace(alathe) = 12 .
  eq : neededspace(anotherlathe) = 15 .
  eq : neededspace(yetanotherlathe) = 11 .
      :
endobj
```

図 4.10

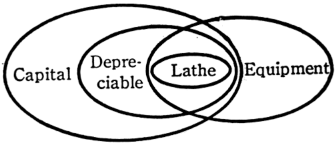


図 4.11　ソートの包含関係の例(2)

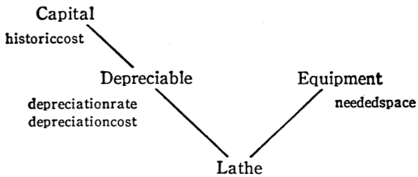


図 4.12　演算の継承の例

順次継承される．たとえば，ソート Lathe（旋盤）は直上のソート Depreciable（減価償却財）から関数 depreciationrate, depreciationcost を，ソート Equipment（設備）から関数 neededspace をそれぞれ継承し，また Depreciable を介してソート Capital（資本）から関数 historiccost を継承する（図 4.12）．ここに抽出した等式は，ソート Lathe の定数 alathe などにおけるこれらの関数の値を定義している．

以上の限られた例でも知られるとおり，順序ソート代数は多ソート代数に比べ記述能力が飛躍的に高い．しかしながら（あるいはそれゆえ），その反面で始代数の存在は自明でなく，ソートの半順序および関数の型に関して，ある条件を満たす必要がある[10]．OBJ2 の項書換えによる実行はこのソートの包含関係を考慮して行なわれている．

4.3.7　属性つきパターンマッチング

OBJ2 は項書換えシステムにより実行されるが，書換え規則を選択する際に行なわれるパターンマッチングで，2引数関数の，以下の属性を利用すべく指示することができる．

- 交換則
- 結合則

```
object SET[ELM :: TRIV] is
  protecting BOOL .
  sort Set .
  op phi : -> Set .         --- 空集合
  op omega : -> Set .       --- 全体集合
  op {_} : Elt -> Set .
  op _$_ : Set Set -> Set
       [associative commutative identity: phi] .
       --- 二つの引数集合のどちらか一つにだけ含まれて
       --- いる要素を集めた集合
  op _&_ : Set Set -> Set
       [associative commutative idempotent identity: omega] .
                              --- 共通集合
  op _|_ : Set Set -> Set .   --- 和集合
  op _-_ : Set Set -> Set .   --- 差集合
  op _in_ : Elt Set -> Bool . --- 集合の要素か？
  vars E E' : Elt .
  vars S S' S'' : Set .
  eq Set : S $ S = phi .
  eq Set : { E } & { E' } = if E == E' then { E } else phi fi .
  eq Set : S & phi = phi .
  eq Set : S & (S' $ S'') = (S & S') $ (S & S'') .
  eq Set : S | S' = (S & S') $ S $ S' .
  eq Set : S - S' = S & (S $ S') .
  eq Bool : E in phi = false .
  eq Bool : E in ({ E' } $ S) = if E == E' then true else E in S fi .
endobj
```

図 4.13

- ●ベキ等則 (idempotency)
- ●零元 (identity) の存在

この機能を利用した例を**図 4.13** に示す.

図 4.13 のパラメータ化されたモジュール SET は幾つかの集合関数（演算）を定義している．関数 $, & に付加された [associative…] の宣言はこれらの関数の属性を表わす．これらの宣言により，関数の定義に必要な等式の数が減り，交換則のように陽に書換え規則として表わすことの困難な性質を扱える．たとえば，$ の零元が phi（空集合）であること，交換則が成り立つこと，の宣言がなければ，

eq Set : S $ phi＝S.

eq Set : phi $ S＝S.

の二つの等式を追加する必要がある．単一化の場合には決定可能性に問題が多く，特に結合則とベキ等則のもとでの等号論理においては一階であっても決定不能であるが，パターンマッチングではそうした問題は生じない.

4.3.8 評価戦略の指定

OBJ2には，関数の引数の評価順序を指定する機能がある．たとえば，さきに示したモジュールBOOLには演算

op_if_then_else_fi : Bool ASort ASort -> ASort.

が宣言されているが，この条件判定の関数は，以下のような順に評価することが望ましい．

条件部（第一引数）

本体

値部（条件部の評価により，第二または第三引数）

この順に評価させる指定は，以下のように行なう．

op if_then_else_fi : Bool ASort ASort -> ASort [(1 0)].

中括弧に囲まれたリストの中の自然数は，引数または本体を表わす．0は本体であり，1，2，… はそれぞれ第一，第二，… の引数である．OBJ2の実行は書換えシステムによるから，この評価順の指定は，リデックス取出し戦略を陽に指示することになる．したがって，引数の評価順の指定により，うまい戦略を指示することができれば，通常の名前呼びや値呼びの戦略では得られない正規項を求めることができる．このような，書換え戦略の指定を伴った書換えシステムでも，あまり効率を落とさずにLisp言語にコンパイルできることがわかっている[11]．

4.3.9 モジュール表現

これまで用いたモジュール参照におけるモジュールの指定は，単に

protecting STATES.

protecting SORTING[INT].

のように，特定のモジュール名を引用するか，パラメータ化されたモジュールの実例を引用していた．OBJ2のモジュールの指定法は，実はこれにとどまらず，高度に洗練されている．この指定法を**モジュール表現**と呼ぶが，それは以下の構造を取り得る．

（1） **モジュール積**　参照されるモジュールのソート名や定数名，(記法を含む)関数記号を変更する．たとえば，

STATES * (sort State to Nation, op (the united kingdom) to (anglo-celtic))

という記法は，モジュール STATES におけるソート State を Nation に，the united kingdom という定数を anglo-celtic におのおの変更する．ほかの定数，たとえば，belgium はそのままである．

（2） **モジュール和**　複数のモジュールを統合して新たなモジュールを構成する．たとえば，

STATES＋SORTINT

という記述は，（有用かどうかはともかく）STATES と SORTINT を統合した新しいモジュール（名前はまだない）を構成する．この新しいモジュールでは，統合された各モジュールで宣言，定義されたソート，関数記号が適当な名前変えを行なって宣言，定義されているとみなされ，また，それらのモジュールが参照しているモジュールが重複を除いた上で参照されている，とみなされる．

（3） **多重束縛**　パラメータの束縛は一度に複数行なえる．たとえば，

2TUPLE [INT, INT]

は OBJ2 の組込みモジュール INT と 2TUPLE（ソートの対を操作する）から整数対を操作するモジュールを構成し，

SEQ [SEQ [INT]]

は整数の列の列を構成する．

以上の表現要素を重ね合わせた，複雑なモジュール式の一例を以下に示す．

```
SEQ[SEQ[INT * (op (_+_) to (__+),
               op (_-_) to (__-),
               op (_*_) to (__*),
               op (_quo_) to (__/))]
      * (sort Seq to ASeq,
         op (.) to (period))
          +
       SEQ[INT]
      * (sort Seq to AnotherSeq)]
       * (sort Seq to YetAnotherSeq)
```

4.4 おわりに

代数型言語 OBJ2 は，記述の容易さと意味定義の厳格さを兼ね備えている点で，抽象データ型の記述言語として現時点で最適なものであるといえる．ま

た，その背景にある意味論の性質から，仕様記述言語に適している．特に，記述能力の点では，モジュールのパラメータ化機構と順序ソート機構により，このクラスの言語の中でも類を見ない能力を持っている．さらに，項書換えシステムという簡明で強力な実行メカニズムを有している点も大きな特徴である．これまでの OBJ2 ないしそのファミリーは試験的に使用されるにとどまっているが，グラフィックシステム[12),13)]のほか，構成管理システム，OBJ 自身，ハードウェア検証システムなどの記述例が報告されている．

現在 OBJ2 またはそのファミリーの処理系には以下のようなものがある．

（1） OBJ0, OBJT (UCLA)

（2） OBJ1 (SRI)

（3） OBJ2 (SRI, ETL)

（4） OBJ3 (SRI)

（5） UMIST-OBJ (Univ. of Manchester, Inst. of Sci. and Tech.)

（6） MC-OBJ (Univ. of Milan)

（7） Franz Lisp OBJ2 (Washington State Univ.)

OBJ0, OBJT, OBJ1 は，ある意味でOBJ2の祖先ということができる．しかし，代数型の言語である点を除けば，言語の基本記述単位を object, theory, view の三つに分類している点などで，OBJ2 はまったく新しい言語であるといえる．OBJ3 は，OBJ2 の言語仕様をほぼそのまま引き継いで，主にリダクションに関する研究の成果を取り入れた処理系を新たに作ったものである．UMIST-OBJ, MC-OBJ はともに，OBJ1 をベースにし，Franz Lisp OBJ2 は，OBJ2 をベースにしている．これらは，OBJ1 や OBJ2 の設計開発者たちとは独立に作成された処理系である．

電総研で稼働しており，筆者らが使用している OBJ2 の処理系は，最初の OBJ2 の処理系である MacLisp 版から Common Lisp 版へ改訂し，若干の改良を加えたものである．このインタプリタは高度の対話性を持ち，1) コマンドが階層構造に構成されている，2) コマンド・ガイドが各階層に付随している，などの特徴を持っている．

OBJ2 以外の代数型言語に視野を広げれば，OSI (Open Systems Interconnection) の形式的仕様記述手段として開発された LOTOS[14)] がすでに国際標準になるこ

とが決まり，一部のサークルで熟成を続けてきた代数に基礎を置く，プログラミング・パラダイムが普及しはじめたという状況である．

最後に文献について少し触れておく．本章では抽象データ型の代数的仕様記述法に基づいて設計された OBJ2 言語の入門的な解説を行なった．抽象データ型とその代数的仕様記述に関しては数多くの文献が存在する．抽象データ型のソフトウェア開発における重要性に関しては 1)，その 理論的背景 については 2) または 3)を参照するのがよいであろう．

OBJ2 に関しては本章が唯一の日本語による入門的な解説である． 本章を読んで興味を持たれた読者は 5)，6)，7) などを参照していただきたい．

参考文献

1) 鳥居宏次，二木厚吉，真野芳久：プログラミング方法論の展望，情報処理，Vol. 20, No. 1, Jan. 1979.

2) Goguen, J. A., Thatcher, J. W. and Wagner, E. G.: An initial algebra approach to the specification, correctness, and implementation of abstract data types, *Current trends in programming methodology, Vol. IV: Data Structuring*, R. T. Yeh, Ed., Prentice-Hall, 1978.

3) 稲垣康善，坂部俊樹：抽象データタイプの代数的仕様記述法の基礎（1)-(4)，情報処理，Vol. 25, No. 1, 5, 7, 9, 1984.

4) 二木厚吉，外山芳人：項書き換え型計算モデルとその応用，情報処理，Vol. 24, No. 2, Feb. pp. 133-146, 1983.

5) Futatsugi, K., Goguen, J., Jouannaud, J.-P. and Meseguer, J.: Principles of OBJ2, *Proc. of 12th Symposium on Principles of Programming Languages*, ACM, pp. 52-66, 1985.

6) Futatsugi, K.: An overview of OBJ2, *Proc. of France-Japan A. I. and C. S. Symposium 86*, Institute for New Generation Computer Technology, Japan, 1986.

7) Futatsugi, K., Goguen, J., Meseguer, J. and Okada, K.: Parameterized Programming in OBJ2, *Proc. of 9th Int. Conf. on S. E.*, March, pp. 51-60, 1987.

8) Goguen, J. A.: Higher Order Functions Considered Unnecessary for Higher Order Programming, *Technical Report SRI-CSL-88-1*, SRI International, 1988.

9) 岡田康治，二木厚吉：プロトタイピング技法の形式化の試み，プロトタイピングと要求定義シンポジウム，情報処理学会，Apr. pp. 29-37, 1986.

10) Goguen, J. A., Meseguer, J.: Order-Sorted Algebra I: Partial and Overloaded Operators, Errors and Inheritance, *Internal Report*, SRI International.

11) 戸村 哲，二木厚吉：項書換えシステムから Lisp プログラムへの変換系，日本ソフトウェア科学会プログラム変換合成研究会 PM-86-11, 1986.

12) Duce, D. A., Fielding, E. V. C.: Formal Specification——A Comparison of Two Techniques, *The Computer Journal*, Vol. 30, No. 4, 1987.

13) Nakagawa, A. T., Futatsugi, K., Tomura, S., Shimizu, T.: Algebraic Specification of

Macintosh's QuickDraw Using OBJ2, *Proc. of 10th ICSE*, 1988.

14) ISO-Information Processing System-Open Systems Interconnection—LOTOS—A Formal Description Technique Based on the Temporal Ordering of Observational Behavior, *IS 8807*, 1989.

(ふたつぎ こうきち，なかがわ あたる)

第5章　プログラム代数とFP



富樫　敦

5.1　はじめに

毎年，野球ファンを一喜一憂させる日本シリーズが幕を閉じ，新人ドラフト会議の話題が新聞のスポーツ欄を賑わすころになると，必ずといっていいくらい，今年のノーベル賞授賞者は誰になるかが話題になる．物理学賞はどこそこの○○博士，化学賞は××研究所の△△博士，…と．残念ながら，計算機科学賞という賞はない．Alfred B. Nobel が遺言を託した 19 世紀の末にすれば，計算機は遥か彼方の未来の代物であり，計算機科学は Nobel にとっては思いもよらない分野だったのだろう．では，計算機科学に携わっている研究者は何も貰えないのかというと，そうでもない．計算機科学の分野にもノーベル賞に相当する賞がある．チューリング賞である．現在ある計算機の数学的モデルを考案した Alan M. Turing の業績を称えた賞である．

1977年のチューリング賞は，IBM の John Backus 博士に輝いた．授賞理由の要約はこうである．「彼の Fortran を通しての，実用的高級プログラミング・システムの設計に対する，深淵にして強い影響力を持ち，かつ消えることのない貢献，及びプログラミング言語の仕様のための形式的手続きに関する学術的な論文発表」（文献１）から引用させていただいた）の理由による授賞である．Fortran とそれに関連する業績で授賞された Backus が，晴れの舞台である授賞記念講演では，Fortran の F の字は出て来るものの，Fortran の設計思想とはまったく相反する新しい計算システム（モデル）について話された．関数型プロ

グラミング・システム FP である[1)~3)]．授賞記念講演から10年以上を得た今，少し古い感もあるが，あの独創的な設計思想，簡潔さ，それに他に類を見ない強力な道具立てであるプログラム代数は，今でも学ぶべきところが多いのではないだろうか．関数型プログラミングにはあまり興味がないという人でも，少なくとも計算機システムの研究に従事している人，あるいはこれから従事しようという人にとって，記念講演の最初の部分だけでも読む価値があるし，お読みになることを勧める．FP の出現によって，新しいパラダイムとしての関数型プログラミングが，現在のように発展してきたといっても過言ではない．それほど FP の与えたインパクトは大きかったのである．

この章では，今日の関数型プログラミング，さらには，新しいプログラミング・パラダイムに影響を与えた，プログラミング・システム FP とプログラム代数のさまざまな醍醐味について，最近の動向も踏まえて解説する．5.2 節でプログラミング・システム FP について紹介し，FP の強力な道具立てであるプログラム代数を 5.3 節で述べる．5.4 節では，FP プログラムを λ 計算に持ち込んで意味を解釈するために，純粋なラムダ計算にリスト構造を導入してラムダ計算（λ-calculus）を少し拡張する．FP プログラムの拡張されたラムダ式への翻訳については，5.5 節で触れる．

5.2　プログラミング・システム FP

FP のプログラムは，ラムダ計算[8)] 同様（関数）式と呼ばれる表現式であり，これによって与えられたデータ（オブジェクト）から結果であるデータを返す関数を表現する．ただ違いは，ラムダ計算では関数を引数として取り関数を返す高階関数を自由に扱えるが，FP ではオブジェクトをオブジェクトに変換する一階の関数だけが用いられ，1 引数の関数だけが許される．もう一つの違いは，FP のプログラムには変数がないことである．これによって，関数レベルでのプログラミングや変換，検証が可能となる．

FP には，あらかじめ定まった基本関数から新しい関数を構成するプログラム構成子，あるいは関数形式が用意されていて，これらを用いることでいくらでも複雑なプログラムを作成できる．タイプを考えた場合，プログラム構成子

は関数を引数に取り関数を返す二階の高階関数と見なせる．以下，順を追って FP システムを紹介する．

5.2.1 オブジェクト

FP のデータは**オブジェクト**（object）と呼ばれ，アトムと列からなる．**アトム**（atom）としては，整数（…, −2, −1, 0, 1, 2, …），ブール値（true, false），文字列（たとえば，"FP", "Backus"）などがある．**列**（sequence）は，アトムあるいは列を要素として一列に並べ，括弧＜と＞で囲んだ記号列である．正確には，オブジェクトは次のようにして帰納的に定義される．

（a） アトムはオブジェクトである．

（b） 未定義，あるいはエラーを表わす要素**ボトム**（bottom）⊥は，オブジェクトである．

（c） $x_1, \cdots, x_n$ がボトムでないオブジェクトならば，これらから作られる列 $<x_1, \cdots, x_n>$ は，オブジェクトである．$x_1, \cdots, x_n$ のうち一つでもボトムがあれば，作られる列は⊥とする．すなわち，

$$<x_1, \cdots, \perp, \cdots, x_n> = \perp.$$

（d） 空列＜＞は，オブジェクトである．

プログラムである関数 f をオブジェクト x に**作用**（apply）することを $f : x$ と書く．FP においては，すべての関数は⊥を保存する．

$$f : \perp = \perp$$

⊥は未定義，あるいはエラーを表わすのであったから，この式は，意味のないものからは意味のないものしか出て来ないことを表わす．⊥を保存する関数は**ストリクト**（strict）と呼ばれる．FP の関数はすべてストリクトである．

FP の関数は，基本関数，プログラム構成子から作られる関数，等式を用いて定義される関数の 3 種類である．

5.2.2 基本関数

FP には最初から備え付けの関数があり，**基本関数**（primitive function）と呼ばれている．代表的な基本関数を少し紹介する．

（a）　選択関数 (selector function)

選択関数は，列からある特定の要素を取り出す関数である．正の整数 **1**，**2**，… は選択関数を表わし，**k**：x は x が長さ k 以上の列のときは k 番目の要素を表わし，それ以外の場合はボトムとなる．

$$\mathbf{2} : <a, b, c, d> = b$$

$$\mathbf{5} : <a, b, c, d> = \bot$$

オブジェクトの正の整数と選択関数とを区別するために，選択関数を太字で表わす（たとえば，**2** のように）．

（b）　列操作関数 (sequence manipulator function)

列操作関数は，列に対して種々の操作を行なうための基本関数である．列操作関数として，FP にはいくつかの強力な関数が用意されている．

tl：列の最初の要素を取り除いた残りの列を値として返す．ただし，列でないオブジェクトや空列に作用したときは⊥を返す．(tail)

$$\mathrm{tl} : <1, 2, 3> = <2, 3>$$

$$\mathrm{tl} : <> = \bot$$

$$\mathrm{tl} : 1 = \bot$$

apndl：列の左側から新たな要素を追加する．(append left)

apndr：列の右側から新たな要素を追加する．(append right)

$$\mathrm{apndl} : <a, <x_1, \cdots, x_n>> = <a, x_1, \cdots, x_n> \qquad (n \geqq 0)$$

$$\mathrm{apndr} : <<x_1, \cdots, x_n>, b> = <x_1, \cdots, x_n, b> \qquad (n \geqq 0)$$

特に $n = 0$ の場合，上式はそれぞれ

$$\mathrm{apndl} : <a, <>> = <a>$$

$$\mathrm{apndr} : <<>, b> = <b>$$

を意味する．

distl：要素を左側から分配する．(distribute from left)

distr：要素を右側から分配する．(distribute from right)

$$\mathrm{distl} : <a, <x_1, \cdots, x_n>> = <<a, x_1>, \cdots, <a, x_n>> \qquad (n \geqq 0)$$

$$\mathrm{distr} : <<x_1, \cdots, x_n>, b> = <<x_1, b>, \cdots, <x_n, b>> \quad (n \geqq 0)$$

apndl, apndr 同様，特に $n = 0$ の場合，上式はそれぞれ

$$\mathrm{distl} : <a, <>> = <>$$

distr：$\langle\langle\rangle, b\rangle = \langle\rangle$

を意味する.

（c）　**列生成関数** (sequence generator function)

iota：1から与えられた正の整数までの整数列を返す.

(number sequence generator)

iota：$n = \langle 1, \cdots, n\rangle$　　$(n \geqq 1)$

iota：$\langle\rangle = \bot$

（d）　**恒等関数** (identity function)

id：与えられたオブジェクトをそのまま返す. (identity)

id：$x = x$

（e）　**算術関数** (arithmetic function)

算術関数として，整数上の四則演算 add（足し算），sub（引き算），mult（掛け算），div（割り算）や suc（後者関数），pred（前者関数）などがある.

add：$\langle 5, 2\rangle = 7$

mult：$\langle 3\rangle = \bot$

div：$\langle 7, 3\rangle = 2$　　（小数点以下を切り捨てる）

pred：$3 = 2$

（f）　**論理関数** (logical function)

論理関数としては，and, or, not などがあり，それぞれ論理積，論理和，否定をとる関数である.

and：$\langle$true, false$\rangle$ = false

not：$\langle 2, 3\rangle = \bot$

（g）　**述語関数** (predicate function)

述語関数は，データの性質をチェックする関数である．オブジェクトの一致を調べる eq (test for equality), 整数 0 であるかどうかを調べる eq0 (test for zero), 数の大小を比較する gt (greater than)，ge (greater than or equal)，lt (less than), le (less than or equal), アトムであるかどうかを確かめる atom (test for atom) や空列のときだけ true を返す null (test for null) などの述語関数が用意されている.

eq：$\langle\langle a, 3\rangle, \langle a, 3\rangle\rangle$ = true

eq：$\langle a, 3\rangle$ = false

$\mathrm{eq0} : 0 = \mathrm{true}$

$\mathrm{gt} : <3, 2> = \mathrm{true}$

$\mathrm{lt} : <3, 2> = \mathrm{false}$

$\mathrm{atom} : \text{“a”} = \mathrm{true}$

$\mathrm{atom} : \bot = \bot$

$\mathrm{null} : <> = \mathrm{true}$

$\mathrm{null} : <a, 3> = \mathrm{false}$

すべての基本関数に共通なことは，意図しないデータ（⊥も含む）に作用すると⊥を返すことである．

5.2.3　プログラム構成子

FPでは，基本関数を次の**プログラム構成子**（program forming operator）を用いて次々に結合し，プログラムである関数を作る．

（**a**）　**関数合成**（composition）：$f \circ g$

$$(f \circ g) : x = f : (g : x)$$

（**b**）　**関数列構成**（construction）：$[f_1, \cdots, f_n]$

$$[f_1, \cdots, f_n] : x = <f_1 : x, \cdots, f_n : x> \qquad (n \geqq 1)$$

（**c**）　**条件式**（condition）：$p \to f \,; g$

$$(p \to f \,; g) : x = \begin{cases} f : x & \text{if } p : x = \mathrm{true} \\ g : x & \text{if } p : x = \mathrm{false} \\ \bot & \text{otherwise} \end{cases}$$

（**d**）　**定数関数化**（constant）：$\bar{x}$

$$\bar{x} : y = \begin{cases} x & \text{if } y \neq \bot \\ \bot & \text{otherwise} \end{cases}$$

（**e**）　**並列作用**（apply to all）：αf

$$\alpha f : x = \begin{cases} <f : x_1, \cdots, f : x_n> & \text{if } x = <x_1, \cdots, x_n> \qquad (n \geqq 0) \\ \bot & \text{otherwise} \end{cases}$$

特に，xが空列，すなわち $n = 0$，のときは

$$\alpha f : <> = <>$$

となることに注意されたい．

（f） **右挿入** (right insert)：$/f$

$$
\begin{aligned}
/f: x = & \; x_1 && \text{if } x = \langle x_1 \rangle \\
& f: \langle x_1, /f: \langle x_2, \cdots, x_n \rangle \rangle && \text{if } x = \langle x_1, \cdots, x_n \rangle \quad (n \geqq 2) \\
& \bot && \text{otherwise}
\end{aligned}
$$

挿入に関しては，（f）の右挿入のほかに双対な左挿入 $/_L f$ があり，

$$
\begin{aligned}
/_L f: x = & \; x_1 && \text{if } x = \langle x_1 \rangle \\
& f: \langle /_L f: \langle x_1, \cdots, x_{n-1} \rangle, x_n \rangle && \text{if } x = \langle x_1, \cdots, x_n \rangle \quad (n \geqq 2) \\
& \bot && \text{otherwise}
\end{aligned}
$$

で定義される．

5.2.4 関数定義

これまで，基本関数からプログラム構成子を用いて次々に関数が結合され，複雑な関数が作られていく様子を見た．FP での実際のプログラミングは，プログラムである関数を定義することで行なわれる．**関数定義**は，次の形をした定義式である．

$$Def\ f = E(f)$$

ここで，f はこれから定義しようとする関数名であり，$E(f)$ は基本関数と定義関数（f を含む）からプログラム構成子を用いて作られる関数式である．$E(f)$ に f が含まれているときは，f に関する再帰的定義式となる．

関数定義の例として，データが数の 0 であるかどうかを確かめる述語関数 eq0 を定義してみると，

$$Def\ \mathrm{eq0} = \mathrm{eq} \circ [\mathrm{id}, \bar{0}]$$

となる．実際，数 0 に適用してみると，

$$
\begin{aligned}
\mathrm{eq0}: 0 &= (\mathrm{eq} \circ [\mathrm{id}, \bar{0}]): 0 \\
&= \mathrm{eq}: ([\mathrm{id}, \bar{0}]: 0) \\
&= \mathrm{eq}: \langle \mathrm{id}: 0, \bar{0}: 0 \rangle \\
&= \mathrm{eq}: \langle 0, 0 \rangle \\
&= \mathrm{true}
\end{aligned}
$$

となり，確かに意図した関数となっている．この例は再帰がない場合であり，この場合の関数定義は関数式の名前づけに対応する．

FPの基本関数には，要素を左側から分配する列操作関数 distl があり，その働きは

$$\text{distl} : \langle a, \langle x_1, \cdots, x_n \rangle\rangle$$
$$= \langle\langle a, x_1\rangle, \cdots, \langle a, x_n\rangle\rangle$$

であった．もう少し複雑な再帰による関数定義の例を見るために，この distl を定義してみよう．$f\times g$ を $[f\circ 1, g\circ 2]$ の略記表現とすると，

$$f\times g : \langle x, y\rangle = \langle f : x, g : y\rangle$$

が成り立つ，distl はこの略記表現を用いて，

$$\textit{Def}\ \text{distl} = \text{null}\circ 2 \rightarrow \overline{\langle\rangle};$$
$$\text{apndl}\circ[\text{id}\times 1, \text{distl}\circ(\text{id}\times \text{tl})]$$

と定義される．引数の2番目の要素について，列の長さに関する帰納法を用いてこの定義を確かめてみよう．列 $\langle a, \langle\rangle\rangle$ に作用した場合には，

$$\text{null}\circ 2 : \langle a, \langle\rangle\rangle = \text{true}$$

となって条件式の最初の関数が作用され，

$$\text{distl} : \langle a, \langle\rangle\rangle$$
$$= \overline{\langle\rangle} : \langle a, \langle\rangle\rangle$$
$$= \langle\rangle.$$

一方，列 $\langle a, \langle x_1, \cdots, x_n\rangle\rangle$, $n\geqq 1$，に作用した場合には，

$$\text{null}\circ 2 : \langle a, \langle x_1, \cdots, x_n\rangle\rangle = \text{false}$$

となり条件式の最後の関数が作用され，

$$\text{distl} : \langle a, \langle x_1, \cdots, x_n\rangle\rangle$$
$$= \text{apndl}\circ[\text{id}\times 1, \text{distl}\circ(\text{id}\times\text{tl})] : \langle a, \langle x_1, \cdots, x_n\rangle\rangle$$
$$= \text{apndl} : \langle\langle \text{id} : a, 1 : \langle x_1, \cdots, x_n\rangle\rangle,$$
$$\text{distl} : \langle \text{id} : a, \text{tl} : \langle x_1, \cdots, x_n\rangle\rangle\rangle$$
$$= \text{apndl} : \langle\langle a, x_1\rangle, \text{distl} : \langle a, \langle x_2, \cdots, x_n\rangle\rangle\rangle.$$

ここで，$\langle x_2, \cdots, x_n\rangle$ は $\langle x_1, \cdots, x_n\rangle$ より長さが1だけ短くなっているので，帰納法の仮定より

$$\text{distl} : \langle a, \langle x_2, \cdots, x_n\rangle\rangle = \langle\langle a, x_2\rangle, \cdots, \langle a, x_n\rangle\rangle$$

が成り立つと仮定すると，

$$\text{distl} : \langle a, \langle x_1, \cdots, x_n\rangle\rangle$$

$= \text{apndl} : \langle\langle a, x_1\rangle, \langle\langle a, x_2\rangle, \cdots, \langle a, x_n\rangle\rangle\rangle$

$= \langle\langle a, x_1\rangle, \cdots, \langle a, x_n\rangle\rangle$

となって，期待どおりの結果を得た.

練習問題 1　基本関数 distr, iota, さらには次の等式

$\text{length} : \langle x_1, \cdots, x_n\rangle = n \qquad (n \geqq 0)$

$\text{reverse} : \langle x_1, \cdots, x_n\rangle = \langle x_n, \cdots, x_1\rangle \qquad (n \geqq 0)$

$\text{append} \langle\langle x_1, \cdots, x_n\rangle, \langle y_1, \cdots, y_m\rangle\rangle$

$= \langle x_1, \cdots, x_n, y_1, \cdots, y_m\rangle \qquad (n, m \geqq 0)$

で表わされる関数 length, reverse, append の定義式を与えよ.

FP の関数（プログラム）を正確に定義すると，次のようにまとめることができる.

（1）　基本関数は，FP の関数である.

（2）　$f_1, \cdots, f_n$ を FP の関数，C を n 個の関数を引数とするプログラム構成子とするとき，C を $f_1, \cdots, f_n$ に適用して構成される関数 $(C, f_1, \cdots, f_n)$ は，FP の関数である.

（3）　関数記号 f を基本関数と見なして構成された FP の関数を $E(f)$ とすると，定義式

$Def\ f = E(f)$

によって（再帰的に）定義される関数 f は，FP の関数である.

5.2.5 FP のプログラミング・スタイル

FP のプログラミング・スタイルの一つは，プログラム構成子を使って比較的単純な関数から複雑な関数を組み上げる，いわゆる関数レベルのプログラミングである．もう一つは，本来は列を再帰的に分解して処理しなければならない操作が，強力な基本関数 distl, distr やプログラム構成子である並列作用 (αf)，右挿入 $(/f)$ により，再帰を用いないで直接的にしかも簡潔に表現できることである．そこで，FP らしいプログラムとは何かを探るため，5.2.4 項の練習問題 1 で取り上げた列の長さを求める関数 length を再考する.

通常，length は次のように再帰的に定義される（たぶん，読者の答えも同じだったと思うが).

Def length ＝ null → $\bar{0}$; suc ∘ length ∘ tl

これでは，列の要素を一度で処理できる α や / を使っていない．FP のプログラミング・スタイルである，関数レベルのプログラミングと強力な基本関数やプログラム構成子の使用を促進するためには，次のようなプログラミング方法論が好ましい．関数はオブジェクトからオブジェクトへの写像であるから，最初に意図するオブジェクトの変換系列を考え，次に一連の変換を行なうための関数を当てはめ，最後にこれらを合成する．ただし，個々の関数にはできるだけ強力な基本関数やプログラム構成子 α，/ を用いるようにする．

length は意図するオブジェクトが列であるから，列の長さを求めるためには，次の一連のオブジェクトの変換を経て所望とする結果を得る．

$<x_1, \cdots, x_n>$

$\mapsto <1, \cdots, 1>$　　（n 個の 1 からなる列）

$\mapsto 1 + \cdots + 1$　　（n 個の 1 の足し算）

$\mapsto n$

最初の変換は $\alpha\bar{1}$ で，要素どうしの演算操作は右挿入（/）の得意とするところだから，$<1, \cdots, 1>$ から $1 + \cdots + 1$ の計算結果である n までの一連の変換は /add で達成できる．後は合成するだけだから，FP らしい length は，

Def length ＝ /add ∘ $\alpha\bar{1}$

と定義される．

練習問題 2　上で定義した length の定義を確かめよ．次に，練習問題 1 の reverse, append さらに列の最後の要素を取り出す関数 last

last：$<x_1, \cdots, x_n> = x_n$　　$(n \geqq 1)$

の再帰的でない（FP らしい）定義式を与えよ．

奨励すべき FP のプログラミング・スタイルは，次のようにまとめることができる．

（a）　与えられた問題を解くための一連のオブジェクトの変換系列を考え，これに着目して問題を比較的小さい問題へ分解する．

（b）　隣り合うオブジェクト間の変換を行なわせるための関数を，強力な関数とプログラム構成子を用いてできるだけ再帰がないように書き下す．

（c） 書き下した関数を合成する.

（d） 最後に, 最適化が可能ならば, （次節で述べる）プログラム代数則を用いて最適化を行なう.

この節を締めくくるに当たり, 今までの総復習を兼ねて, 要素がアトムからなる有限集合の直積集合 $A \times B$ を求めるプログラムを考えてみよう. 集合を列で表わすことにすると, 求めるプログラムは, たとえば二つの集合の対

$$<<a, b, c>, <1, 2>> \qquad \text{(obj-1)}$$

で作用したとき, 集合

$$<<a, 1>, <a, 2>, <b, 1>, <b, 2>, <c, 1>, <c, 2>> \qquad \text{(obj-4)}$$

を生成する関数 cross である. ここで, cross は基本関数 distl, あるいは distr を一般化した関数であることに注意されたい. (obj-1) から (obj-4) を得ることは, 次の変換系列によって達成できる.

(obj-1)

$$\mapsto <<a, <1, 2>>, <b, <1, 2>>, <c, <1, 2>>> \qquad \text{(obj-2)}$$

$$\mapsto <<<a, 1>, <a, 2>>, <<b, 1>, <b, 2>>, <<c, 1>, <c, 2>>> \qquad \text{(obj-3)}$$

$\mapsto$ (obj-4)

(obj-1) から (obj-2) への変換は, 基本関数 distr で行なえる. また, たとえば(obj-2) の最初の要素である $<a, <1, 2>>$ に distl を作用させれば $<<a, 1>, <a, 2>>$ を得るから, この操作を列の（トップレベルの）すべての要素に行なうようにして (obj-2) に α distl を作用させれば, (obj-3) を得る. 後は, たとえば

$$\text{concat} : <<3, 4>, <d, e>, <5, 6>> = <3, 4, d, e, 5, 6>$$

となる concat を (obj-3) に作用させれば (obj-4) を得るから, cross は最終的には,

Def cross $=$ concat $\circ$ α distl $\circ$ distr

で定義される. 後残るのは concat の定義であるが, 実は

Def concat $=$ /(/apndl $\circ$ apndr)

で定義される. 詳細については読者にゆだねたい. ヒントとして, 二つの列をこの順番で結合する append

$$\text{append} : \langle\langle 3,4\rangle, \langle d,e\rangle\rangle = \langle 3,4,d,e\rangle$$

の定義が

$$Def\ \text{append} = /\text{apndl} \circ \text{apndr}$$

となるのを確かめた後，concat の定義を確認されたい．

練習問題 3　concat を確認し，cross の定義を完結せよ．さらに，$A \times B$を求めるプログラムを発展させて，より一般的な問題である有限集合 A_i の直積集合 $A_1 \times \cdots \times A_n$ を求めるプログラムを考えよ（問題の解法を通して，FP の簡潔さを，そしてプログラム構成子の強力さを体感するものと確信する）．

5.3　プログラム代数

「プログラム代数」なんて，FP にあまりなじみのない人には聞き慣れない言葉である．「ブール代数とか群や環だったら知っている」という人が多いかもしれない．この節では，本章のタイトルの一部でもある FP に特有なプログラム代数の醍醐味を披露したい．

5.3.1　プログラム代数則

FP では，プログラムである関数に代数構造を見いだすことができる．言いかえれば，プログラムをある数学的な対象とし，プログラム構成子を代数演算とする代数系，**プログラム代数** (program algebra) と呼ぶ，を構築できることである．このプログラム代数に対しても，一般の代数系と同じようにある等式，**プログラム代数則** (algebraic law)，が成り立つ．これらは，プログラム構成子や基本関数の定義から導出できる等式である．代表的な代数則を少し挙げてみよう．

プログラム代数則

(A1)　$h \circ (p \rightarrow f;\ g) = p \rightarrow h \circ f;\ h \circ g$

(A2)　$(p \rightarrow f;\ g) \circ h = p \circ h \rightarrow f \circ h;\ g \circ h$

(A3)　$/f \circ [g_1, \cdots, g_n] = f \circ [g_1, /f \circ [g_2, \cdots, g_n]]$　　$(n \geqq 2)$

(A4)　$/f \circ [g] = g$

(A 3*) $/_L f \circ [g_1, \cdots, g_n]$

$= f \circ [/_L f \circ [g_1, \cdots, g_{n-1}], g_n]$ $(n \geqq 2)$

(A 4*) $/_L f \circ [g] = g$

(A 5) $[f, g] \circ h = [f \circ h,\ g \circ h]$

(A 6) $g \to\to 1 \circ [f, g] = f$

(A 7) $f \to\to 2 \circ [f, g] = g$

(A 8) $p \to (p \to f\ ;\ g)\ ;\ h = p \to f\ ;\ h$

(A 9) $\alpha f \circ \mathrm{apndl} \circ [g, h] = \mathrm{apndl} \circ [f \circ g, \alpha f \circ h]$

(A 9*) $\alpha f \circ \mathrm{apndr} \circ [g, h] = \mathrm{apndr} \circ [\alpha f \circ g, f \circ h]$

(A 10) $/f \circ \mathrm{apndl} \circ [g, h] = f \circ [g, /f \circ h]$

(A 10*) $/_L f \circ \mathrm{apndr} \circ [g, h] = f \circ [/_L f \circ g, h]$

(A 11) $f \to\to \bar{x} \circ f = \bar{x}$

ここで，等式 $f=g$ は，f と g が関数として等しいこと，すなわち任意のオブジェクト x について，

$f : x = g : x$

が成り立つことである．また，$p \to\to f = g$ は，関数 p が定義されている領域（$p : x \neq \bot$ となる x の集合であり，p の**定義域** (domain) と呼ばれる）で f と g が等しいことを表わす．また，＊の付いた代数則は，同じ番号の代数則と双対をなす代数則である．

ここで挙げた以外にも有用な代数則がある．幾つか挙げてみよう．

(A 5)，(A 6)，(A 7) の一般化として

(A 5') $[f_1, \cdots, f_n] \circ g = [f_1 \circ g, \cdots, f_n \circ g]$ $(n \geqq 1)$

(A 6') $\mathrm{seq} \circ g \to\to 1 \circ \mathrm{apndl} \circ [f, g] = f$

(A 7') $f \to\to \mathbf{i} \circ \mathrm{apndl} \circ [f, g] = \mathbf{i-1} \circ g$ $(i \geqq 2)$

ここで，seq は列であるかどうかを判定する述語関数であり，

Def seq = null → true ; seq ∘ tl

で定義される．

関数合成と並列構成に関して

(A 12) $\alpha f \circ [g_1, \cdots, g_n] = [f \circ g_1, \cdots, f \circ g_n]$ $(n \geqq 1)$

(A 13) $\mathrm{distl} \circ [f, [g_1, \cdots, g_n]] = [[f,\ g_1], \cdots, [f, g_n]]$ $(n \geqq 1)$

(A 13*) $\mathrm{distr} \circ [[g_1, \cdots, g_n], h] = [[g_1, h], \cdots, [g_n, h]]$ $(n \geqq 1)$

(A14)　$\text{apndl} \circ [f, [g_1, \cdots, g_n]] = [f, g_1, \cdots, g_n]$　　$(n \geqq 1)$

(A14*)　$\text{apndr} \circ [[g_1, \cdots, g_n], h] = [g_1, \cdots, g_n, h]$　　$(n \geqq 1)$

条件式と並列構成に関して

(A15)　$[f_1, \cdots, (p \to g\,;\, h), \cdots, f_n]$

$$= p \to [f_1, \cdots, g, \cdots, f_n]\,;\, [f_1, \cdots, h, \cdots, f_n] \qquad (n \geqq 1)$$

並列作用と関数合成に関して

(A16)　$\alpha(f \circ g) = (\alpha f \circ \alpha g)$

これら以外にも有用な代数則を挙げることができる．(A12) は形こそ違え，代数則としては (A9) と等価である．自分なりにこれぞというのを見つけてみるのも興味深い．

5.3.2　代数則の証明

ほとんどの代数則は自明であろう．中には場合分けを必要とする規則もあるが，証明の中心は単純な式の書換えである．たとえば，一見難しそうに見える代数則 (A12) も，以下に示すように簡単な式の書換えで事足りる．

$$\begin{aligned}
&\alpha f \circ [g_1, \cdots, g_n] : x \\
&\quad = \alpha f : ([g_1, \cdots, g_n] : x) \\
&\quad = \alpha f : \langle g_1 : x, \cdots, g_n : x \rangle \\
&\quad = \langle f : (g_1 : x), \cdots, f : (g_n : x) \rangle \\
&\quad = \langle f \circ g_1 : x, \cdots, f \circ g_n : x \rangle \\
&\quad = [f \circ g_1, \cdots, f \circ g_n] : x
\end{aligned}$$

そのほかの代数則の証明は読者にゆだねたい．証明を通して FP の基本関数やプログラム構成子に慣れてほしい．

練習問題 4　5.3.1項で挙げたプログラム代数則を証明せよ．

5.3.3　代数則とプログラム

プログラムの性質を確かめたり，プログラム変換を行なったりする場合，5.3.2項で代数則を証明したようにオブジェクト・レベルまで遡ってもできるが，それでは代数則を苦労して調べた意味がない．そこで，簡単ではあるが，プログラムの性質を証明したりプログラムを簡単化するのに，代数則がどのくらい貢献するかを見てみよう．

集合の直積を求めるときに，複数の列を順序よく結合して一本の列にまとめる補助関数 concat を用いた．定義は

$$Def\ \text{concat} = /(/\text{apndl} \circ \text{apndr})$$

である．この concat について少し性質を調べてみよう．最初の性質は，並列作用構成子（α）との関係である．

$$\alpha f \circ \text{concat} = \text{concat} \circ \alpha(\alpha f) \qquad \text{(prop)}$$

この性質は，直感的にいうと，列の列 $<S_1, \cdots, S_n>$ を作用の対象とし，各列 S_i をこの順番で結合してできた列 S の各要素に並列に関数 f を作用することは，各 S_i のそれぞれの要素に f を並列に作用させ，得られた列をこの順番で結合することに等しいことを主張している．それでは，この性質を代数則を用いて正確に示してみよう．

等式 $f=g$ が意味するところは，f と g の定義域が同じで，しかも，定義域中のどのオブジェクトに作用しても，得られる結果は同じということである．(prop) の両辺が意図する定義域は複数個の列からなる列であり，代数則を用いて示すためには，このようなオブジェクトを関数レベルまで引き上げる必要がある．

$$[[f_{11}, \cdots, f_{1n_1}], [f_{21}, \cdots, f_{2n_2}], \cdots, [f_{m1}, \cdots, f_{mn_m}]] : x$$
$$= << x_{11}, \cdots, x_{1n_1}>, <x_{21}, \cdots, x_{2n_2}>, \cdots, <x_{m1}, \cdots, x_{mn_m}>>$$

であるから，(prop) を示すためには，次式を示すことで十分．ここで，$x_{ij} = f_{ij} : x$ と置いた（$1 \leqq i \leqq m$, $1 \leqq j \leqq n_i$）．

$$\alpha f \circ \text{concat} \circ [[f_{11}, \cdots, f_{1n_1}], \cdots, [f_{m1}, \cdots, f_{mn_m}]]$$
$$= \text{concat} \circ \alpha(\alpha f) \circ [[f_{11}, \cdots, f_{1n_1}], \cdots, [f_{m1}, \cdots, f_{mn_m}]]$$

代数則を用いて実際に示すと，次のようになる．

$$\alpha f \circ \text{concat} \circ [[f_{11}, \cdots, f_{1n_1}], \cdots, [f_{m1}, \cdots, f_{mn_m}]]$$
$$= \alpha f \circ [f_{11}, \cdots, f_{1n_1}, \cdots, f_{m1}, \cdots, f_{mn_m}] \qquad \text{(concat の定義から)}$$
$$= [f \circ f_{11}, \cdots, f \circ f_{1n_1}, \cdots, f \circ f_{m1}, \cdots, f \circ f_{mn_m}] \qquad \text{(代数則 (A12) から)}$$
$$= \text{concat} \circ [[f \circ f_{11}, \cdots, f \circ f_{1n_1}], \cdots, [f \circ f_{m1}, \cdots, f \circ f_{mn_m}]]$$

（再び concat の定義から）

$$= \text{concat} \circ [\alpha f \circ [f_{11}, \cdots, f_{1n_1}], \cdots, \alpha f \circ [f_{m1}, \cdots, f_{mn_m}]]$$

（代数則（A12）から）

$$= \text{concat} \circ \alpha(\alpha f) \circ [[f_{11}, \cdots, f_{1n_1}], \cdots, [f_{m1}, \cdots, f_{mn_m}]]$$

（代数則 (A12) から）

したがって，(prop) が成り立つ．

ただし，上の証明で concat の定義から直ちに

$$\text{concat} \circ [[f_{11}, \cdots, f_{1n_1}], \cdots, [f_{m1}, \cdots, f_{mn_m}]]$$
$$= [f_{11}, \cdots, f_{1n_1}, \cdots, f_{m1}, \cdots, f_{mn_m}]$$

が成り立つとしたが，正確にはconcatの定義に基づいて（帰納法により）正確に証明しなければならない．これも読者への練習問題としたい．同様にして，concatに関する性質

$$/f \circ \text{concat} = /f \circ \alpha(/f)$$
$$/\text{add} \circ \alpha(\text{length}) = \text{length} \circ \text{concat}$$

を示すことができる．ただし，f は結合律が成り立つ関数であるとする．

練習問題5　上記の concat に関する性質を代数則を用いて証明せよ．

プログラム代数則は，単にプログラムの性質を調べるためだけに用いられるわけではない．プログラム変換やプログラムの簡単化にも使用される[14]．5.2.5項では，FP のプログラミング・スタイルに着目して，通常再帰的に定義されるlengthについて，再帰のない定義を得た．実は，このような再帰的定義式から再帰のない定義式へのプログラムの変換（あるいは，導出）が，代数則と線形展開定理を駆使してほぼ機械的に行なうことができる．**線形展開定理** (linear expansion theorem) は Backus によって与えられ，再帰的に定義された関数が線形性を満たせば，再帰のない無限展開式に直すことができるという定理である（ただし，有限的に表現できるとは限らない）[1]．紙面の都合で立ち入った解説はできないが，興味のある読者は文献1)，2)，14) を参照していただきたい．Backus のオリジナルな文献1) よりは，後で時間をかけて整理した文献2) のほうが読みやすい．また，文献7) の5章では，例を豊富に取り入れてこの話題について説明している．特に，再帰的に定義されたlengthの定義から，再帰のない定義を代数則と線形展開定理を用いて機械的に得る方法に興味のある方は，是非この文献を読まれたい．ただ初学者にとっては，この辺の話題は少し重荷であろう．最初は，話題の項目程度に止めておき，だいぶ慣れてきた時期に細部まで読まれることを勧める．

5.4　ラムダ計算の拡張

FP のプログラムを実際に実行させるためには，5.2 節で述べた言語仕様に従ってプログラムを解釈する処理系が必要となる．従来，FP プログラムの操作的意味を与える方法として，FP を形式化した**形式システム FFP**(Formal system for FP)，さらには，関数適用を効率良く行なうために FFP に状態を持ち込んだ**作用的状態遷移システム AST**(Applicative State Transition system) が採用され，FP プログラムはすべて FFP あるいは AST の表現に翻訳されて解釈された[1),13)]．FP にはオブジェクト，オブジェクトからオブジェクトへの写像である関数，関数から関数への写像であるプログラム構成子と暗黙のタイプがあるが，FFP や ASTはタイプのないシステムである．FP のプログラムは，いったんタイプのない世界に持ち込まれ，変換された表現に対する操作によって元のプログラムの（操作的）意味が与えられる．このような意味の与え方は，別に FP に特有な方法ではない．関数型言語 Miranda や ML でも，特に再帰を扱うために，いったんタイプのないラムダ計算[8)]に持込み意味を与えている[7),9)]．

ここでは，FP プログラムをFFP あるいはAST の表現に翻訳して解釈する代わりに，関数型プログラムの代表的な計算モデルであるラムダ計算に持ち込んで，プログラムの意味を解釈したい．ただ残念なことに，ラムダ計算は変数束縛（ラムダ抽象）と関数適用の機能だけを備えた比較的単純な計算モデルであり，FP のオブジェクトである列や関数列をラムダ計算で自然に表現するためには，ラムダ計算にリスト構造が不可欠となる．そこで，FP プログラムの具体的な翻訳の仕方や解釈は次節に譲り，本節ではリスト構造が自然に扱えるようにするために，ラムダ計算を少し拡張する．

5.4.1　ラムダ式の拡張

タイプ付きラムダ計算に基礎を置く Miranda や ML，パラダイムを超えて Horn 節論理に基づく Prolog でもそうであるが†，たいていのプログラミング言

† Miranda や ML，Prolog については文献 10) のそれぞれ 7 章，6 章および 3 章で最近の動向も考慮して解説している．

語ではリストは単なるデータ構造として扱われ，プログラムがリストとして表現されることはない．ただし，例外もある．LISP では，データだけでなく，プログラムもリスト構造として表現される．プログラムとデータを統一的な枠組みで扱えることの意義は大きい．これから紹介する拡張されたラムダ計算では，関数を要素とするリストがまた関数であり，プログラムである関数とデータがリスト構造の中に統一的に埋め込まれる．

読者の中には，「わざわざリスト構造をラムダ計算に持ち込まなくとも，純粋なラムダ計算の枠組みでリストやリスト処理関数が定義可能なのでは？」と指摘される方もおられよう．しかし，この場合のリスト処理はラムダ計算の β 簡約を介して行なわれ，あまり直接的ではない．より直接的でしかも簡潔な方法は，リスト表記を許し，リスト処理関数を定数にしてその働きを簡約規則として与える方法である．

以下に，新たに拡張されたラムダ式の構文規則を BNF（ちなみに，BNF の B はBackus の頭文字である）で与える．

拡張されたラムダ式の定義

＜ラムダ式＞::=＜変数＞|＜定数＞|＜ラムダ抽象＞|＜関数適用＞|
　　　　　　＜リスト＞

＜ラムダ抽象＞::=ラムダ＜変数＞.＜ラムダ式＞

＜関数適用＞::=＜ラムダ式＞＜ラムダ式＞

＜リスト＞::=[＜ラムダ式＞＜リスト末尾＞|[]

＜リスト末尾＞::=,＜ラムダ式＞＜リスト末尾＞|]

表 5.1　代表的な定数の例

定数の種類	具体例
整数	…, −2, 1, 0, 1, 2, …
文字列	“FP”, “Backus”, …
未定義記号	⊥
射影関数	1, 2, …
算術演算子	**+, −, *, /, suc, pred**
論理演算子	∧, ∨, ¬
述語	**≒, >, ≧, <, ≦, atom, null**
リスト操作用演算子	**head, tail, cons, map, append**
コンビネータ	**true, false, Y, I, …**

ここで，＜定数＞としては，**表 5.1** で掲げた定数などがあるとする．また慣例に従い，変数として x, y, z などの英小文字を使用する．

ラムダ式の例

$(\lambda y.[x, y, z])\ 3$

$(\lambda x.[xy, yx])M$

$(\lambda x.(\lambda y.[x, y])a)[b, c, d]$

などが具体的な拡張されたラムダ式の例である．これら拡張されたラムダ式にも通常のラムダ式のように β 簡約規則を定義し，それぞれ $[x, 3, z]$, $[My, yM]$, $[[b, c, d], a]$ に簡約できるようにしたい．この節では，ラムダ計算における基本的概念や慣用的な表記法については，読者はある程度承知しているものとして解説している．多少不慣れな読者には，『新しいプログラミング・パラダイム[10)]』第2章「ラムダ計算と高階プログラミング」や文献8) を読まれることを勧める．特にラムダ抽象と関数適用に関しては以下の略記法を採用している．

$$\lambda x_1 \cdots x_n . M \equiv (\lambda x_1 .(\lambda x_2 .(\cdots(\lambda x_n . M)\cdots)))$$

$$MN_1N_2\cdots N_n \equiv ((\cdots((MN_1)N_2)\cdots)N_n)$$

すなわち，ラムダ抽象については右結合を，関数適用については左結合を仮定する．以下，簡単化のため，特に断らない限り，ラムダ式，ラムダ計算といった場合には，拡張された意味でのラムダ式，ラムダ計算を指すものとする．

5.4.2 拡張されたラムダ計算の簡約規則

以下，拡張に基づいて計算モデルとしてのラムダ計算の簡約規則を列挙していく．個々の規則は矢印(→)をはさんで左辺と右辺からなり，左辺の形をしたラムダ式は，対応する右辺の形をしたラムダ式に書換えができることを意味する．

(a) α 規則

束縛変数の名前の違いを除いて同じラムダ式を同一視し，あえて規則は明示しない．たとえば，$\lambda x.[x, (\lambda y. yx)]$ の束縛変数 x, y をそれぞれ u, v で置き換えると $\lambda u.[u, (\lambda v. vu)]$ を得るから，$\lambda x.[x, (\lambda y. yx)]$ と $\lambda u.[u, (\lambda v. vu)]$ は同じラムダ式である．

(b) β 簡約規則

$$(\lambda x. M)N \rightarrow [N/x]M$$

ここで，$[N/x]M$ はM中の自由変数xをすべてNで置き換えたラムダ式を表わす．ただし，N中の自由変数が $[N/x]M$ で束縛変数にならないように，M中の束縛変数の名前を組織的に付け換える．

α 規則（明示的に与えていないが）と β 簡約規則は，本来のラムダ計算の α 規則や β 簡約規則と何ら変わるところがない．リストと関数適用およびラムダ抽象との関係を定めたのが次の γ 簡約規則である．

（c）　γ 簡約規則

（γ1）　$[M_1, \cdots, M_n]N \to [M_1N, \cdots, M_nN]$　　（$n \geqq 0$）

（γ2）　$\lambda x.[M_1, \cdots, M_n] \to [\lambda x.M_1, \cdots, \lambda x.M_n]$　　（$n \geqq 0$）

$n=0$ の場合に相当する$[\,]N$ と $\lambda x.[\,]$は，ともに$[\,]$に簡約されることに注意されたい．引き続き，リスト操作用演算子，射影関数，コンビネータに関する簡約規則を示す．意味は規則から自明であろう．

（d）　リスト操作用演算子に関する簡約規則

head $[M_1, \cdots, M_n] \to M_1$　　（$n \geqq 1$）

tail $[\,] \to [\,]$

tail $[M_1, \cdots, M_n] \to [M_2, \cdots, M_n]$　　（$n \geqq 1$）

cons $A[M_1, \cdots, M_n] \to [A, M_1, \cdots, M_n]$　　（$n \geqq 0$）

ここで，**cons** $A[M_1, \cdots, M_n]$ は括弧を省略しない (**cons** A)$[M_1, \cdots, M_n]$ の略記表現であった．また，**head** は $[\,]$ に対しては未定義であり，**head** $[\,]$ は簡約できないことに注意されたい．

（e）　射影関数に関する簡約規則

1 $[M_1, \cdots, M_n] \to M_1$　　（$n \geqq 1$）

k $[M_1, \cdots, M_n] \to$ **pred k** $[M_2, \cdots, M_n]$　　（**k**>1，$n \geqq 1$）

（f）　コンビネータに関する簡約規則

true $M\,N \to M$

false $M\,N \to N$

Y$M \to M$(**Y**M)

I $M \to M$

以下同様に，算術演算子，論理演算子，述語に関しても簡約規則が定義されているとする．たとえば，

$+\ 2\ 3 \to 5$

pred 4 → 3

pred 3 → 4

∧ **true false** → **false**

≐ [a] [] → **false**

atom 2 → **true**

などのように．射影関数に対しても算術演算子が定義されていることに注意されたい．たとえば，4 は射影関数であり，**pred** 4 は簡約されて射影関数の 3 に書き換えられる．ラムダ計算全体を通して，2引数以上の演算子や述語は**カリー化** (Currying) されているものとする．

述語の中には，リストが空であるかどうかを調べる **null** があり，その規則は以下で与えられる．

null [] → **true**

null $[M_1, \cdots, M_n]$ → **false**　　$(n \geqq 1)$

また，FP の述語関数 eq0 に相当する述語 **eq0** は，述語 ≐ により

eq0 $= \lambda x. \doteq x\ 0$

と定義できる．

ここで注意することは，簡約規則を用いて定義した演算子や関数が，すべてのデータに対しては必ずしも定義されていないことである．たとえば，(**null** 3) は簡約不能であり，未定義であることを表わす定数，たとえば⊥，にはあえて書き換えない．このようなラムダ式は，もうこれ以上簡約できない**正規形** (normal form) として放っておかれる．場合によっては，意味のない式が意味のある式の部分式として現われ，簡約の過程で消えてしまうことがある．

2 ([∧, +]3 4)

⇒ 2 ([∧3, +3]4)

⇒ 2 [∧3 4, +3 4]

⇒ **pred** 2 [+3 4]

⇒ 1 [+3 4]

⇒ +3 4

⇒ 7

もう一つ例を挙げる．A, Bを適当なラムダ式とすると 2(**cons** A(($\lambda xy.$ **cons** xy)

$B[\,]))$ は

$$2\,(\mathbf{cons}\,A((\lambda xy.\,\mathbf{cons}\,xy)B[\,]))$$
$$\Rightarrow 2\,(\mathbf{cons}\,A((\lambda y.\,\mathbf{cons}\,B\,y)[\,]))$$
$$\Rightarrow 2\,(\mathbf{cons}\,A(\mathbf{cons}\,B[\,]))$$
$$\Rightarrow 2\,(\mathbf{cons}\,A[B])$$
$$\Rightarrow 2[A,B]$$
$$\Rightarrow \mathbf{pred}\,2[B]$$
$$\Rightarrow 1[B]$$
$$\Rightarrow B$$

とBに簡約され，さらにBの正規形まで簡約される．一方，$3[A,B]$は正規形である$(1[\,])$に簡約され，もうこれ以上評価されない．たとえば，$2[\wedge 3\,4, +3\,4]$ のように簡約できる式，**リデックス**（redex）と呼ぶ，が二つ以上ある場合，どのリデックスを優先して簡約するかを決めた，計算に関するメタな規則を**計算戦略**（computation strategy）と呼ぶ．計算戦略によっては，求まる正規形も求まらない場合がある．ここでは，計算戦略として，リデックスが複数あった場合には，最も左に位置するリデックスを書き換える，**正規化戦略**（normalizing strategy）†である，**最左戦略**（left-most strategy）を採用する．Mに簡約規則を1回（0回以上）適用してNが得られることを $M\Rightarrow N(M\overset{*}{\Rightarrow}N)$ で表わす．

ラムダ計算の簡約規則にもう少し慣れるために，FPのプログラム構成子である関数合成（$\circ$）を適当なラムダ式で表現し，簡単なプログラム代数

$$[f,g]\circ h=[f\circ h,g\circ h] \tag{A5}$$
$$f\circ(g\circ h)=(f\circ g)\circ h \tag{assoc}$$

を確かめてみよう．そこで，コンビネータ**B**

$$\mathbf{B}=\lambda uvx.\,u(vx)$$

により，$f\circ g$ を

$$f\circ g=\mathbf{B}fg$$

で定義する．

代数則（A5）について確かめる．はじめに，左辺を簡約すると

$$[f,g]\circ h$$

† 正規化戦略とは，正規形が存在するならば，必ずこの正規形を求めることができる戦略である．

$= (\lambda uvx.\,u(vx))[f, g]\,h$ (関数合成の定義から)

$\stackrel{*}{\Rightarrow} \lambda x.[f, g](hx)$ (β簡約規則を 2 回適用して)

$\Rightarrow \lambda x.[f(hx), g(hx)]$ ((γ1) より)

$\Rightarrow [\lambda x.\,f(hx), \lambda x.\,g(hx)]$ ((γ2) より)

となる．続いて，右辺を簡約すると

$[f \circ h, g \circ h]$

$= [(\lambda uvx.\,u(vx))f\,h, (\lambda uvx.\,u(vx))g\,h]$ (関数合成の定義から)

$\stackrel{*}{\Rightarrow} [\lambda x.\,f(hx), \lambda x.\,g(hx)]$ (β簡約規則をそれぞれ 2 回適用して)

となり，両辺が同じラムダ式に簡約できたので，確かに成り立つ．

一方，(assoc) については，

$f \circ (g \circ h)$

$= (\lambda uvx.\,u(vx))f(\mathbf{B}\,gh)$ (関数合成の定義から)

$\stackrel{*}{\Rightarrow} \lambda x.\,f(\mathbf{B}\,ghx)$ (β簡約規則を 2 回適用して)

$= \lambda x.\,f((\lambda uvy.\,u(vy))ghx)$ (関数合成の定義から)

$\stackrel{*}{\Rightarrow} \lambda x.\,f(g(hx))$ (β簡約規則を 3 回適用して)

$(f \circ g) \circ h$

$= (\lambda uvx.\,u(vx))(\mathbf{B}fg)h$ (関数合成の定義から)

$\stackrel{*}{\Rightarrow} \lambda x.\,\mathbf{B}fg(hx)$ (β簡約規則を 2 回適用して)

$= \lambda x.\,(\lambda uvy.\,u(vy))fg(hx)$ (関数合成の定義から)

$\stackrel{*}{\Rightarrow} \lambda x.\,f(g(hx))$ (β簡約規則を 3 回適用して)

とやはり両辺が同じラムダ式に簡約でき，成り立つことが確かめられた．

5.4.3 そのほかのリスト操作用演算子

5.4.2 項で定義した以外のそのほかのリスト操作用演算子が，今まで述べた基本演算子を用いてどのように定義できるかを少し見てみたい．

cons は要素 A とリスト $[M_1, \cdots, M_n]$ を引数に取り，A をリストの先頭に加えたリスト $[A, M_1, \cdots, M_n]$ を返す関数であった．**cons** と少し似た働きをする二つのリストを与えられた順番で結合する演算子 **append** を定義してみよう．

$$\textbf{append}\,[M_1, \cdots, M_m]\,[N_1, \cdots, N_n] = [M_1, \cdots, M_m, N_1, \cdots, N_n]$$

リスト操作用基本演算子を用いて再帰的に定義すると，

$$\mathbf{append}\,xy = \text{if}\,(\mathbf{null}\,x)\,y$$
$$\text{else}\,(\mathbf{cons}\,(\mathbf{head}\,x)\,(\mathbf{append}\,(\mathbf{tail}\,x)\,y))$$

と書ける.上式をラムダ抽象して,**シンタックス・シュガリング** (syntax sugaring) を行なわないで正式なラムダ式として書き直すと,

$$\mathbf{append} = \lambda xy.(\mathbf{null}\,x)\,y\,(\mathbf{cons}\,(\mathbf{head}\,x)\,(\mathbf{append}\,(\mathbf{tail}\,x)\,y))$$

となる.これは,**append** に関する再帰的定義であり,**不動点コンビネータ** (fixed point combinator) **Y** により解を求めると,**append** の正確な定義を得る.

$$\mathbf{append} = \mathbf{Y}\,(\lambda fxy.(\mathbf{null}\,x)\,y\,(\mathbf{cons}\,(\mathbf{head}\,x)\,(f\,(\mathbf{tail}\,x)\,y)))$$

以上で定義が終わった.これが想定した演算子になっているかどうかを確かめるために,**append**$[a,b,c][d,e]$ を計算してみると,期待した計算結果$[a,b,c,d,e]$ を得る.

以後,**append** の定義として,簡約規則による次の定義を採用する.

$$\mathbf{append}\,[N_1,\cdots,N_m]\,[M_1,\cdots,M_n]$$
$$\to [N_1,\cdots,N_m,M_1,\cdots,M_n] \qquad (m,\ n\geqq 0)$$

実際,この規則はmに関する帰納法で示すことができる.

もう一つの重要な演算子は,**map** 演算子である.関数とリストを引数に取り,関数をリストの各要素にそれぞれ作用した結果のリストを返す.

$$\mathbf{map}\,F\,[M_1,\cdots,M_n] = [FM_1,\cdots,FM_n]$$

map も **append** 同様再帰的に定義され,

$$\mathbf{map}\,fx = \text{if}\,(\mathbf{null}\,x)\,[\,]$$
$$\text{else}\,(\mathbf{cons}\,(f\,(\mathbf{head}\,x))\,(\mathbf{map}\,f\,(\mathbf{tail}\,x))),$$

Y コンビネータにより,正確な定義

$$\mathbf{map} = \mathbf{Y}\,(\lambda mfx.(\mathbf{null}\,x)\,[\,]$$
$$(\mathbf{cons}\,(f\,(\mathbf{head}\,x))\,(m\ \ f\,(\mathbf{tail}\,x))))$$

を得る.これより,**map** に関する簡約規則を求めると

$$\mathbf{map}\,F\,[M_1,\cdots,M_n] \to [FM_1,\cdots,FM_n] \qquad (n\geqq 0)$$

を得る.

練習問題6　**append**$[a,b,c][d,e]$ を計算すると $[a,b,c,d,e]$ となることを確認せよ.さらに,**append** および **map** のラムダ式による定義から,簡約規則によるそれぞれの定義を確かめよ.

5.4.4 相互再帰的定義式

ラムダ計算にリスト構造を持ち込んだことによって，**相互再帰的定義式**（mutual recursive definition）を巧妙に扱うことができる．次のように複数の等式によって，相互に再帰的に定義される関数 $f_1, \cdots, f_n$ を考えてみる．

$$f_1 = E_1$$
$$\vdots$$
$$f_n = E_n$$

ここで，f_i は関数の名前を意味する変数であり，E_i はラムダ式である．いま，$f_1, \cdots, f_n$ を自由変数として含むラムダ式 E を評価してみよう．もし f_i が E_j, $j > i$，以降だけに出現するならば，E は

$$(\lambda f_n \cdots f_2 f_1.\ E)\, E_n \cdots E_2 E_1$$

を計算することで評価できる．相互再帰のように，f_i, f_j がそれぞれ E_j, E_i に自由に現われるような場合は，こう簡単にはいかない．ただし，自己再帰だけで相互再帰がない場合は，定義式の順番を適当に定めることで処理できる．ただし，$f_i = E_i$ が自己再帰の場合は，f_i に E_i の代わりに $\mathbf{Y}(\lambda f_i.\ E_i)$ を代入する．

ラムダ式を拡張したことにより，相互再帰が巧妙に扱えることを例で見てみよう．互いの関数を相互に呼び合う整数上の関数を g, h とする．

$$g\ n = \mathrm{if}\,(\doteq n\ 0)\ \mathrm{then}\ 0\ \mathrm{else}\,(+\,(g\,(\mathbf{pred}\ n))\,(h\,(\mathbf{pred}\ n)))$$
$$h\ n = \mathrm{if}\,(\doteq n\ 0)\ \mathrm{then}\ 1\ \mathrm{else}\,(*\,(g\ n)\,(h\,(\mathbf{pred}\ n)))$$

変数を除去し，より正確に表記すると次を得る．

$$g = \lambda n.(\doteq n\ 0)\,0\,(+\,(g\,(\mathbf{pred}\ n))\,(h\,(\mathbf{pred}\ n)))$$
$$h = \lambda n.(\doteq n\ 0)\,1\,(*\,(g\ n)\,(h\,(\mathbf{pred}\ n)))$$

$[g, h]$ を表わす新しい変数 f を導入し，

$$f = [g, h]$$

と置くと，g, h は

$$g = \mathbf{1} f$$
$$h = \mathbf{2} f$$

と表わされる．そこで，g, h の定義式を f を用いて書き換える．

$$\mathbf{1}\,f = \lambda n.(\doteq n\ 0)\,0\,(+\,(\mathbf{1}\,f\,(\mathbf{pred}\ n))\,(\mathbf{2}\,f\,(\mathbf{pred}\ n)))$$
$$\mathbf{2}\,f = \lambda n.(\doteq n\ 0)\,1\,(*\,(\mathbf{1}\,f\ n)\,(\mathbf{2}\,f\,(\mathbf{pred}\ n)))$$

さらに，二つの定義式を結合すると，

$$f = [\lambda n.(\doteq n\,0)\,0\,(+\,(1\,f(\mathbf{pred}\,n))\,(2\,f(\mathbf{pred}\,n))),$$
$$\lambda n.(\doteq n\,0)\,1\,(*\,(1\,f\,n)\,(2\,f(\mathbf{pred}\,n)))]$$

を得る．この式は，二つの相互再帰的定義式をリスト表記した式であり，その解は**Y**コンビネータにより，

$$f = \mathbf{Y}(\lambda f.[\lambda n.(\doteq n\,0)\,0\,(+\,(1\,f(\mathbf{pred}\,n))\,(2\,f(\mathbf{pred}\,n))),$$
$$\lambda n.(\doteq n\,0)\,1\,(*\,(1\,f\,n)\,(2\,f(\mathbf{pred}\,n)))])$$

と求まる．この解によって，たとえば $(g\,3)$ は，$(1\,f\,3)$ を規則に従って簡約することで評価できる．

練習問題7　実際に $(g\,3)$ を評価し，リスト表記による相互再帰的定義式の取扱いを確認せよ．さらに，上の例を参考にして，一般の場合について考察せよ．

5.4.5 ストリーム（無限リスト）処理

ストリーム（stream），あるいは，**無限リスト**（infinite list）とも呼ぶ，を使うことによって，プログラミングの世界が大幅に広がる．本書1章と9章や文献10）の7章でもストリームに触れた話題がいくつかある．お読みになれば，ストリームの醍醐味を十分満喫されることと思う．実は，ストリームは再帰によって実現でき，処理する場合は**要求駆動**（call by need）† で評価すればよい（幸いにも最左戦略は要求駆動による評価法に一致する）．たとえば，自然数のストリームは，次式で定義される *nat*

$$nat = \mathbf{Y}(\lambda sn.\,\mathbf{cons}\,n\,(s\,(\mathbf{suc}\,n)))$$

により，$(nat\,0)$ で実現できる．実際，

$$S = \lambda sn.\,\mathbf{cons}\,n\,(s\,(\mathbf{suc}\,n))$$

と置くと，

$$\begin{aligned}&nat\,0\\ &= \mathbf{Y}S\,0\\ &\Rightarrow S(\mathbf{Y}S)\,0\\ &\Rightarrow \mathbf{cons}\,0\,((\mathbf{Y}S)\,(\mathbf{suc}\;0))\end{aligned}$$

---

† 要求駆動で評価するとは，要求が生じた段階で評価を進めることである．最も外側に位置するリデックスを優先的に計算する，たとえば，最左戦略や**並列最外戦略**（parallel outer-most strategy）で評価することにほぼ匹敵する．

$\Rightarrow$ **cons** 0 (S (**Y**S) (**suc** 0))

$\Rightarrow$ **cons** 0 (**cons** (**suc** 0) ((**Y**S) (**suc** (**suc** 0))))

$\Rightarrow$ **cons** 0 (**cons** 1 ((**Y**S) (**suc** (**suc** 0))))

と次々とストリームが成長していく．ただ困ったことに，**cons** に関する規則がいつまで経っても適用できないので，たとえば5番目の要素を取り出すために 5 (*nat* 0) を評価しても依然としてストリームを作り続け，どんなに待っても期待した4は返ってこない．これでは使い物にならない．ところが，幸いにも，リストの表記法と幾つかの簡約規則を書き換えるだけで，実は物になるのである．では，どんな修正が必要となるのだろうか．リストの定義については，Prolog でお馴染みの表記法〔_|_〕を採用して，

＜リスト＞::=〔＜ラムダ式＞|＜リスト＞〕|〔〕

とし，たとえば〔a, b, c〕は〔a |〔b |〔c |〔〕〕〕〕の略記とする．α, β 規則，算術演算子，論理演算子，コンビネータや **null** を除いた述語に関する規則については前のままでよいが，そのほかの規則については，次のように修正しなければならない．

γ 規則について

〔〕$N \to$〔〕

〔$A|M$〕$N \to$〔$AN|MN$〕

$\lambda x.$〔〕$\to$〔〕

$\lambda x.$〔$A|M$〕$\to$〔$\lambda x. A|\lambda x. M$〕

リスト操作用演算子について

head〔$A|M$〕$\to A$

tail〔〕$\to$〔〕

tail〔$A|M$〕$\to M$

cons $A\ M \to$〔$A|M$〕

射影関数について

1〔$A|M$〕$\to A$

k〔$A|M$〕$\to$ (**pred k**) M

append, map について

append〔〕$M \to M$

append〔$A|M$〕$N \to$〔A|**append** $M\ N$〕

map F $[\,] \to [\,]$

map F $[A|M] \to [FA|$**map** $F\,M]$

述語について

null $[\,] \to$ **true**

null $[A|M] \to$ **false**

5.5　FPプログラムの翻訳

いよいよ，FPのプログラムをラムダ式に翻訳し，ラムダ計算の中でプログラムを解釈してみよう．変換されたFPの式が，すでに定義した簡約規則により正規形になるまで簡約され，この正規形が元の式の計算結果となる．ラムダ計算に持ち込む利点は，FPのプログラムを解釈できることだけではない．実は，FPの代数則を簡約規則を用いて示すことができるのである．この話題については，この節の最後で触れる．

しかし，ここで一つだけ問題がある．FPにおいては，すべての関数（および列構成子<…>）はストリクト，引数の中に一つでも未定義なものがあれば返す値も未定義，であった．一方，ラムダ計算の関数（および，リスト構成子[_|_]）のほとんどはストリクトではない．したがって，議論の対象となるFPはストリクトでない版とする．

5.5.1　オブジェクトの翻訳

FPのアトムはラムダ計算の定数であり，列はリストで表現する．**表5.2**に，FPのオブジェクトと対応するラムダ式（ラムダ計算への翻訳表現）の対応表

表5.2　オブジェクトの翻訳

FPのオブジェクト	対応するラムダ式
アトム：	定数：
整数（…, −2, −1, 0, 1, 2, …）	整数（…, −2, −1, 0, 1, 2, …）
ブール値（true, false）	コンビネータ（**true**, **false**）
文字列（“FP”, “Backus”）	文字列（“FP”, “Backus”）
未定義記号（⊥）	⊥
列：$<x_1, \cdots, x_n>$	リスト：$[X_1, \cdots, X_n]$

を示す（表の中で，X_i はオブジェクト x_i のラムダ計算への翻訳によるラムダ式を表わす）．たとえば，

$<3, \perp, <a, b>, \mathrm{true}>$ ☞ $[3, \perp, [a, b], \mathbf{true}]$

のように変換する．

5.5.2 基本関数の翻訳

FP の選択関数は，すでに射影関数としてラムダ計算にある．そのほかの基本関数については，以下の対応を考える．

tl ☞ **tail**

apndl ☞ **cons**

id ☞ $\mathbf{I} = \lambda x. x$

算術関数，論理関数や述語関数に関しては，対応する演算子や述語が（定数として扱われるが）すでにラムダ計算の中に用意されている．ただし，多引数の関数はカリー化して，１引数に変換する必要がある．たとえば，数の大小を比較する基本関数 ge に対応する述語 **ge** は，述語 ≧ により

$\mathbf{ge} = \lambda x. \geqq (1x)(2x)$

のようにカリー化されて定義され，これをリスト $[5, 3]$ に適用すると，簡約規則により次のように変形される．

$\mathbf{ge}\ [5, 3] = (\lambda x. \geqq (1x)(2x))[5, 3]$

$\Rightarrow \geqq (1\,[5, 3])(2\,[5, 3]))$

$\Rightarrow \geqq 5\ 3$

$\Rightarrow$ **true**

そのほかのいくつかの関数についての対応を示すと，次のようになる．

add ☞ $\mathbf{add} = \lambda x. + (1x)(2x)$

pred ☞ **pred**

and ☞ $\mathbf{and} = \lambda x. \wedge (1x)(2x)$

eq0 ☞ $\mathbf{eq0} = \lambda x. \doteq x\ 0$

null ☞ **null**

基本関数 iota に対応する関数 **iota** は，次のように再帰的に表現され，

$\mathbf{iota}\ n = \mathrm{if}\ n \doteq 1\ \mathrm{then}\ [1]\ \mathrm{else}\ (\mathbf{append}(\mathbf{iota}(\mathbf{pred}\ n))[n])$

コンビネータ **Y** を用いて，次のように正確に定義される．

表 5.3　基本関数の翻訳

FP の基本関数	対応するラムダ式
選択関数：1, 2, …	射影関数：1, 2, …
列操作関数： tl, apndl, apndr, distl, distr	リスト操作用演算子： **tail, cons, apndr, distl, distr**
列生成関数：iota	関数：**iota**
恒等関数：id	コンビネータ：**I**
算術関数： add, sub, mult, div, suc, pred	算術演算子： **add, sub, mult, div, suc, pred**
論理関数：and, or, not	論理演算子：**and, or, ¬**
述語関数： eq, eq0, gt, ge, lt, le, atom, null	述語： **eq, eq0, gt, ge, lt, le, atom, null**

$$\mathbf{iota} = \mathbf{Y}(\lambda fn.(\doteq n\,1)[1](\mathbf{append}(f(\mathbf{pred}\,n))[n]))$$

同様にして，distl に対応する **distl** は，

$$\mathbf{distl} = \mathbf{Y}(\lambda fax.(\mathbf{null}\,x)[\,](\mathbf{cons}\,[a, (\mathbf{head}\,x)](fa(\mathbf{tail}\,x))))$$

で与えられる．基本関数からラムダ式への翻訳をまとめると，**表5.3** のようになる．

練習問題8　FP の基本関数 apndr, distr に対応するリスト操作用演算子 **apndr, distr** をラムダ式で定義せよ．

練習問題9　個々の基本関数の翻訳を確認せよ．

5.5.3　プログラム構成子の翻訳

プログラム構成子 C と関数 $f_1, \cdots, f_n$ によって，新しい関数 $(C, f_1, \cdots, f_n)$ が作られる．プログラム構成子が，ラムダ計算へどのように翻訳されるかを見てみよう．以下では，FP の関数と変換されたラムダ計算の関数（ラムダ式）を区別するために，FP の関数を小文字（たとえば f）で表わし，対応する変換されたラムダ式を大文字（たとえば F）で表わす．

（a）　関数合成：$f \circ g$

関数合成 $(f \circ g)$ については，コンビネータ

$$\mathbf{B} = \lambda fgx.f(gx)$$

を用いて，次のように対応をとる．

$f \circ g$ ☞ $\mathbf{B}FG$

$f \circ g$ をオブジェクト x に作用することは $((f \circ g) : x)$，ラムダ式に変換された

関数 $(\mathbf{B}\,F\,G)$ を x に対応するラムダ式 X に関数適用することになる $((\mathbf{B}\,F\,G)\,X)$.
実際適用してみると,

$$(\mathbf{B}\,F\,G)\,X = (\lambda fgx.\,f(gx))\,F\,G\,X$$
$$\overset{*}{\Rightarrow} F(GX)$$

となり, 確かに $(\mathbf{B}\,F\,G)$ は関数合成 $(f\circ g)$ に対応することが確かめられた. 以下, 残りのプログラム構成子の翻訳の仕方を列挙すると, 次のようになる.

（b） 関数列構成：$[f_1, \cdots, f_n]$

$[f_1, \cdots, f_n]$ ☞ $[F_1, \cdots, F_n]$

（c） 条件式：$p \to f\,;\,g$

コンビネータ **cond**

$$\mathbf{cond} = \lambda pfgx.\,(px)\,(fx)\,(gx)$$

により, 条件式は式のように変換される.

$p \to f\,;\,g$ ☞ $\mathbf{cond}\;P\,F\,G$

（d） 定数関数化：$\bar{x}$

$\bar{x}$ ☞ $\lambda z.X$

（e） 並列作用：αf

αf ☞ $\mathbf{map}\;F$

（f） 右挿入：$/f$

コンビネータ **insert**

$$\mathbf{insert} = \mathbf{Y}\,(\lambda ins\,fx.\,(\mathbf{null}\,(\mathbf{tail}\;x))\,(\mathbf{head}\;x)$$
$$(f(\mathbf{head}\;x)\,(ins\,f(\mathbf{tail}\;x))))$$

により, 右挿入は次のように変換される.

$/f$ ☞ $\mathbf{insert}\;F$

プログラム構成子の翻訳の仕方をまとめると, **表 5.4** のようになる.

表 5.4 プログラム構成子の翻訳

FP のプログラム構成子	対応するラムダ式
関数合成：$(f\circ g)$	$\mathbf{B}\,F\,G$
関数列構成：$[f_1, \cdots, f_n]$	$[F_1, \cdots, F_n]$
条件式：$(p \to f;\ g)$	$\mathbf{cond}\;P\,F\,G$
定数関数化：$\bar{x}$	$\lambda z.\,X$
並列作用：αf	$\mathbf{map}\;F$
右挿入：$/f$	$\mathbf{insert}\;F$

練習問題10　FP のプログラム構成子が，忠実にラムダ式に変換されていることを確認せよ．

5.5.4　簡約規則を用いた代数則の検証

最後に，ラムダ計算の簡約規則を用いて 5.3.2 項で示したプログラム代数則（A12）を証明してみよう．

対応規則より，

$\alpha f \circ [g_1, \cdots, g_n]$ ☞ $\mathbf{B}(\mathbf{map}\ F)[G_1, \cdots, G_n]$

$[f \circ g_1, \cdots, f \circ g_n]$ ☞ $[\mathbf{B}\ F\ G_1, \cdots, \mathbf{B}\ F\ G_n]$

を得る．両辺を簡約すると，

$\mathbf{B}(\mathbf{map}\ F)[G_1, \cdots, G_n]$

$\overset{*}{\Rightarrow} \lambda X.(\mathbf{map}\ F)([G_1, \cdots, G_n]\ X)$　（Bの定義より）

$\Rightarrow \lambda X.(\mathbf{map}\ F)[G_1 X, \cdots, G_n X]$　（（γ1）の適用）

$\Rightarrow \lambda X.[F(G_1 X), \cdots, F(G_n X)]$　（map に関する簡約規則の適用）

$\Rightarrow [\lambda X.F(G_1 X), \cdots, \lambda X.F(G_n X)]$　（（γ2）の適用）

$[\mathbf{B}\ F\ G_1, \cdots, \mathbf{B}\ F\ G_n]$

$\Rightarrow [\lambda X.F(G_1 X), \cdots, \lambda X.F(G_n X)$　（Bの定義より）

となって証明することができた．そのほかの代数則についても同様に示すことができる．ただし，規則によっては場合分けや帰納法を必要とするときがある．

練習問題11　そのほかの代数則についても，簡約規則を用いて同じように検証してみよ．

5.6　おわりに

ここまで読み進めてこられて，読者はきっとプログラミング・システム FP に十分堪能されたことと思う．特に，プログラムに代数構造を見つけ，プログラムの変換や最適化，さらには検証に利用しようというアイディアはかなり斬新である．また，ラムダ計算にリスト構造を導入して，拡張されたラムダ計算の中で FP のプログラムを解釈できたり，プログラム代数則を証明できることは興味深い．この章を閉じるに当たって，少し参考文献に触れてみたい．

FPについての原典は，チューリング賞授賞記念講演の内容を公にした文献 1)であるが，後続の2)，3)のほうが読みやすい．中でも，2)は本章で触れることができなかった線形展開定理や，非線形の定義の展開定理まで触れていて，かなり読みごたえがある．また，関数型プログラミングについて書かれた本 7)，9)，11)は，1章を割いてFPについてわかりやすく解説している．本章を書く上で参考にした．FPのプログラムをFFPの表現に変換して解釈する方法については，同じIBM勤務のJohn H. Williamsが記した13)がいいだろう．14)はBackusによるプログラム代数をさらに発展させている．Böhm treeの名で有名なイタリアのBöhmは，FPを特別なコンビネータ論理，あるいは数学的な代数系に埋め込み，その中でFPの代数的性質を調べている[5]．Böhmとは違った観点から，GibertはFPを抽象的なコンビネータ代数に埋め込み，FPの基本関数やプログラム構成子を説明している[6]．5.5節で解説したFPのプログラムを，拡張されたラムダ計算に翻訳して解釈する方法は，文献11)によった．そのほか，例題からFPのプログラムを半自動的に合成する方法[4]などの話題もある．これも併せて参考にされたい．

参考文献

1) Backus, J.: Can programming be liberated from the von Neumann style? A functional style and its algebra of programs, *CACM*, Vol. 21, No. 8, pp. 613–641, 1978. (プログラミングはフォン・ノイマン・スタイルから解放されうるか？ 関数的プログラミング・スタイルとそのプログラム代数(1)，(2)，(3)，米澤明憲訳，**bit**，Vol. 11，No. 9～No. 11，共立出版，1979.)

2) Backus, J.: The algebra of functional programs: functional level reasoning, linear equations, and extended definitions, pp. 1–43, *Lecture Notes in Computer Science*, No. 107, Springer-Verlag, 1981.

3) Backus, J.: Function level programs and mathematical objects, *Proc. of the 1981 Conf. on Functional Programming Languages and Computer Architecture*, pp. 1–10, 1981.

4) Banerjee, D.: A methodology for synthesis of recursive functional programs, *ACM Trans. on Prog. Lang. and Sys.*, Vol. 9, No. 3, pp. 441–462, 1987.

5) Böhm, C.: Combinatory foundation of functional programming, *ACM Symp. on Lisp and Functional Programming*, pp. 29–36, 1892.

6) Gibert, J.: Functional programming with combinators, *J. Symbolic Computation*, Vol. 4, pp. 269–293, 1987.

7) Henson, M. C.: *Elements of functional languages*, Blackwell Scientific Pub., 1987.

8) Hindley, J. R., Seldin, J. P.: *Introduction to Combinators and λ-calculus*, Cambridge Univ. Press, 1986.
9) 井田哲雄：『プログラミング言語の新潮流』，共立出版，1988.
10) 井田哲雄編：『新しいプログラミング・パラダイム』，共立出版，1989.
11) Revesz, G. E.: *Lambda-calculus, combinators, and functional programming*, Cambridge University Press, 1988.
12) 富樫　敦: 関数プログラミングにおける計算モデル，情報処理，Vol. 29，No. 8，pp. 817-828, 1988.
13) Williams, J. H.: Notes on the FP style of functional Programming, in *Functional Programming and its Applications*, Darlington, J. et al. eds., Cambridge University Press, pp. 73-101, 1982.
14) Williams, J. H.: On the development of the algebra of functional programs, *ACM Trans. on Prog. Lang. and Sys.*, Vol. 4, No. 4, pp. 733-757, 1982.

（とがし あつし）

第6章 カテゴリカル・プログラミング





横内寛文

6.1 はじめに

カテゴリ (category) という哲学用語とともに，一部の熱狂的愛好家によって本来純粋数学の一分野であったカテゴリ理論が計算機科学に持ち込まれようとしている．カテゴリ理論は，数学の諸分野に共通に表われる原理をまとめようとした試みの集大成である．数学的な対象を定義し，その性質を調べるという数学の基本的な行動には，しばしば共通の手法が見いだされる．カテゴリ理論は，その共通の手法を統一的に説明できる原理を抽出しようと試みている．その結果として，カテゴリ理論は数学的対象を定義したり，その対象の性質を述べるための強力な言語を提供している．プログラムも文字どおりプログラマの意図を表現する言語であり，この点がまさにカテゴリ理論を計算機科学に持ち込もうとする理由である．もちろん，数学とプログラミングは本質的に異なる行動であり，数学で使われる言語がそのままプログラミングで使えるとは思えない．しかし，プログラム，プログラム言語，あるいは計算そのものの性質を調べる上では，カテゴリの考え方が有効になる場合がある．たとえば，プログラム言語の理論と実際の両方の分野で，現在最もホットな話題の一つにポリモルフィズム (polymorphism) がある．ポリモルフィズムとは何かを理論的に解明しようとしている人々の間で，カテゴリの考え方が役に立っている．しかし，この章ではカテゴリの手法と言語をもっと直接的にプログラム言語に利用して，まとまった成果をあげた例を紹介する．

「すべての数学的対象を矢印で記述しよう」カテゴリ理論の最も初歩の段階のスローガンである．少し説明が必要である．数学的な命題を形式的に記述する論理式やプログラム言語のほとんどは，「値」が一番基本の要素である．たとえば，論理式 $(\forall x)$ $(P(x)$ & $Q(x))$ は，すべての値 a について，a は P でありかつ a は Q であることを意味する．すなわち，値を中心にした記述になっている．一方，カテゴリ理論では値を表に出さずに，すべての概念を関数的に記述しようとする．関数という言葉は，すでに集合論の中で値をもとにしたきっちりした定義を持つのでよくない．カテゴリ理論では，関数という言葉の代わりに，もっと抽象的に定義された**矢印**（arrow）ないしは**射**（morphism）と呼ばれる言葉を使う．この矢印を用いて，先ほどの論理式も含めてすべての数学的対象が記述される．

プログラム言語でも論理式と同様に値が基本にある．プログラム言語を，カテゴリの考え方に基づいて矢印だけで記述するように書き換えたら，どういう言語ができ上がるかを調べてみたい．これがこの章の目標である．このためには，プログラム言語を一つ決めなくてはならない．そこで，最も理論的に扱いやすい関数型言語とその基本になるラムダ計算を取り上げる．ラムダ計算とその拡張である関数型言語に対して，カテゴリ流の考え方に従って矢印を基本にした言語を定義する．でき上がった言語は一種のプログラム言語になっているはずである．結論を先に言うと，このプログラム言語は第5章で解説した FP によく似た言語になる．したがって，そのままプログラム言語に使ってもよいが，ここではこの新しい言語をもとにして関数型言語の処理系を設計する．というのも，この言語はそのままで機械語に近い形をしているからである．また，この言語はもとのラムダ計算および関数型言語の意味を記述する言語になっている．

以下，6.2節でカテゴリ流に矢印で記述するための基本的な考え方を具体例を使って解説する．なお，矢印という言葉は誤解を生みそうなので，以後は**モルフィズム**と呼ぶことにする．6.3節では，6.2節の手法をラムダ計算に適用し，ラムダ計算のもう一つの形式化である CCL と呼ぶ体系を導入する．6.4節ではラムダ計算を拡張して簡単な関数型言語 ML/O を定義する．ラムダ計算の拡張に伴って CCL を拡張した言語を定義し，ML/O との違いを調べる．

さらに，拡張 CCL に基づいて ML/O のコンパイラを設計する.

6.2 値からモルフィズムへ

カテゴリ流のモルフィズムを使った表現法を，具体例を使って説明する.今，2引数関数 f と g がすでに定義されているとして，新しく2引数関数 h を次のように定義したとする.

$$h(x, y)=f(x, g(y, x)) \qquad \text{(等式A)}$$

この関数 h をもっと違う形で定義しようと思う.

まず，f と g の定義域を明確にする．一般に，k が集合 X から集合 Y への関数であるとき，

$$k : X \rightarrow Y$$

と書く．2引数以上の関数の定義域は，直積で表わすことにする．たとえば，2引数関数 f の第1引数が集合 A の上を動き，第2引数が集合 B の上を動くとすると，f の定義域は A と B の直積 $A \times B$ となる．直積 $A \times B$ の要素は，A の要素 a と B の要素 b の組 (a, b) で表わされるので，

$$A \times B=\{(a, b) \mid a \in A, b \in B\}$$

となる．関数 f の定義域を直積 $A \times B$ と考えると，$a \in A$, $b \in B$ について，$f(a, b)$ は $f((a, b))$ と書いたほうがはっきりするが，括弧が二重に付くので省略して $f(a, b)$ と書くことにする.

以上の取決めをもとにして，f と g の定義域と値域を

$$f : A \times B \rightarrow C, \quad g : D \times A \rightarrow B$$

と定義すると，h は

$$h : A \times D \rightarrow C$$

となる.

2引数以上の関数を扱うために導入した直積に関して，基本操作をいくつか定義しておく．集合 X と Y について，二つの関数 $\pi_1{}^{X,Y}$ と $\pi_2{}^{X,Y}$ を次のように定義する.

$$\pi_1{}^{X,Y} : X \times Y \rightarrow X, \quad \pi_1{}^{X,Y}(x, y)=x$$

$$\pi_2{}^{X,Y} : X \times Y \rightarrow Y, \quad \pi_2{}^{X,Y}(x, y)=y$$

ふつう $\pi_1{}^{X,Y}$ と $\pi_2{}^{X,Y}$ は射影（projection）と呼ばれている．また，同じ定義域を持つ二つの関数 $k_1 : Z \to X$ と $k_2 : Z \to Y$ について，関数$<k_1, k_2>$を次のように定義する．

$$<k_1, k_2> : Z \to X \times Y$$

$$<k_1, k_2>(z) = (k_1(z), k_2(z))$$

関数の組$<k_1, k_2>$と直積の要素（a, b）の違いに注意されたい．$<k_1, k_2>$ は関数であり，（a, b）は値である．ついでに，関数合成 $\circ$ の定義を明確にしておく．二つの関数 $k_1 : X \to Y$ と $k_2 : Y \to Z$ について，合成関数 $k_2 \circ k_1$ を次のように定義する．

$$k_2 \circ k_1 : X \to Z,\quad (k_2 \circ k_1)(x) = k_2(k_1(x))$$

以上の基本関数および基本操作を用いて，先ほどの関数 h の定義について考える．関数 h の定義式は，関数 f を引数（$x, g(y, x)$）に作用[†]させた形をしている．この引数部分を独立に計算する関数 h_1 を

$$h_1 : A \times D \to A \times B,\quad h_1(x, y) = (x, g(y, x))$$

と定義する．h の定義式は関数合成を用いて，

$$h = f \circ h_1$$

と書き直せる．さらに，関数 h_2 を

$$h_2 : A \times D \to B,\quad h_2(x, y) = g(y, x)$$

と定義すると，h_1 は直積に関する基本関数を用いて，

$$h_1 = <\pi_1{}^{A,D}, h_2>$$

と表わせる．関数 h_2 も同様に

$$h_2 = g \circ <\pi_2{}^{A,D}, \pi_1{}^{A,D}>$$

と表わせる．以上をまとめると，結局 h は

$$h = f \circ <\pi_1{}^{A,D}, g \circ <\pi_2{}^{A,D}, \pi_1{}^{A,D}>> \qquad \text{（等式B）}$$

と表現できることがわかる．

念のために，この式が正しいことを確かめてみる．関数 h の定義域は $A \times D$ だったので，上の等式の右辺を（x, y）$\in A \times D$ に作用させると次のようになる．

$$(f \circ <\pi_1{}^{A,D},\ g \circ <\pi_2{}^{A,D}, \pi_1{}^{A,D}>>)\ (x, y) = f(x, g(y, x))$$

---

† 関数 f と引数 a が与えられていて，$f(a)$ を求めることを「f を a に作用させる」という言い方をする．

この計算は，基本関数 $\pi_1^{A,D}$，$\pi_2^{A,D}$ と関数の基本操作 $<k_1, k_2>, k_2 \circ k_1$ の定義に基づいて機械的に行なえる．この等式は，h を定義している（等式A）を仮定すると（等式B）が導かれ，逆に h が（等式B）で定義されていたと仮定すると（等式A）が導けることを示している．したがって，関数 h の（等式A）による定義と（等式B）による定義は同等であることがわかる．

この（等式B）が，カテゴリ流のモルフィズムを使った定義式である．（等式A）が値に着目した定義式であるのに対して，（等式B）は関数そのものに着目した定義式である点が異なる．（等式B）では，すでに定義されている関数 f, g と基本関数 $\pi_1^{A,D}$，$\pi_2^{A,D}$ をもとに，関数操作 $<k_1, k_2>$ と $k_2 \circ k_1$ を用いて順々に関数を組み上げて，新しい関数 h が定義されている．この点が通常の定義法と違うところである．

実は，カテゴリ理論では上で定義した直積に関する基本操作も含めて，もっと抽象的に，もっと統一的にさまざまな数学的構造が定義される．しかし，この章で必要なカテゴリ流の基本的な考え方は，上の例で十分に示されている．カテゴリの定義を含めて，カテゴリ流定義法の本当の姿はほかのたくさんあるカテゴリ理論の教科書を参照されたい．

6.3 カテゴリカル・コンビネータ

関数の作用，仮引数の機構を抽象化した理論に**ラムダ計算**（λ-calculus）がある．この節では，6.2 節で述べた関数の新しい表現法をもとにしてラムダ計算の意味論を調べ，**カテゴリカル・コンビネータ理論** CCL（categorical combinatory logic）と呼ばれる形式体系を導入する．CCL は，いくつかの基本関数と関数合成 $f \circ g$ などの関数操作から成り立つ言語を持ち，関数間の関係を等式で表わした公理が定義されている．CCL とラムダ計算は本質的に同等の形式体系であるが，ラムダ計算が値に着目した体系であるのに対して，CCL は関数自体を重視した体系である点が異なる．また，CCL は形式体系であると同時にラムダ計算の意味論を定義する代数系ともみなせる．

6.3.1　ラムダ計算の意味論

はじめにラムダ計算の定義を述べておく．この項では特に型付きのラムダ計算を用いる．型の集合を次のように定義する．

（1）　基本型．

（2）　AとBが型のときB^Aも型である．（B^AはAからBへの関数全体を表わす型．）

型付きのラムダ式を次のように定義する．

（1）　変数．（各変数には型が一つ定義されているとする．）

（2）　MがB^Aのラムダ式，Nが型Aのラムダ式のとき，MNは型Bのラムダ式である．MをNに作用させることを意味する．

（3）　xが型Aの変数，Mが型Bのラムダ式のとき，$\lambda x.M$は型B^Aのラムダ式である．ラムダ束縛と呼ばれている．

以上のように形式的に定義された型付きのラムダ式に意味を与える方法を考える．型Aのラムダ式Mは型Aのある値を表わしているはずである．しかし，Mが自由変数†を含んでいる場合には，自由変数の値が決まって初めてMの値が決まる．一般に，自由変数とその値の対応を**環境**（environment）と呼ぶ．たとえば Lisp の処理系を考えるとわかりやすいだろう．

型Aのラムダ式と環境ρに対して，型Aの値$\mathcal{M}[\![M]\!]_\rho$を次のように定義する．

（1）　Mが変数xのとき，

$$\mathcal{M}[\![x]\!]_\rho=\rho(x)$$

ただし，$\rho(x)$は環境ρにおけるxの値を表わす．

（2）　Mが関数作用M_1M_2のとき，

$$\mathcal{M}[\![M_1M_2]\!]_\rho=\mathcal{M}[\![M_1]\!]_\rho(\mathcal{M}[\![M_2]\!]_\rho)$$

M_1の型をB^A, M_2の型をAとすると，$\mathcal{M}[\![M_1]\!]_\rho$は型$B^A$の値，すなわち$A$から$B$への関数$f$を表わす．同様に$\mathcal{M}[\![M_2]\!]_\rho$は型$A$のある値$a$を表わすので，$\mathcal{M}[\![M_1M_2]\!]_\rho$は上の定義のように$f$を$a$に作用させて得られた型$B$の値$f(a)$となる．

（3）　Mがラムダ束縛$\lambda x.M_1$のとき，

---

† 自由変数とは，どのλのスコープにも入らない変数をいう．

$\mathcal{M}[\![\lambda x.M_1]\!]_\rho = k$

ただし，x の型を A, M_1 の型を B として，k は次のように定義した型 B^A の値すなわち A から B への関数を表わす．型 A の値 a について，

$k(a) = \mathcal{M}[\![M_1]\!]_{\rho(x \leftarrow a)}$

ここで，$\rho(x \leftarrow a)$ は環境 ρ 中の変数 x の値を a に置き換えることを表わしている．

上で定義した $\mathcal{M}[\![M]\!]_\rho$ はラムダ式に対するインタプリタと思ってもさしつかえない．この $\mathcal{M}[\![M]\!]_\rho$ でラムダ式の意味が正確に定義される．**表示的意味論**（denotational semantics）の分野では，$\mathcal{M}[\![M]\!]_\rho$ は記号的な式 M に意味を与えていることから，**意味関数**（meaning function）と呼ばれている．

なお，厳密にいうと型 A の値 a というような言い方は正確でない．型も記号の一種だから，型に対しても数学的な意味を与える必要がある．型に集合を割り当てる別の意味関数を定義して，型 A に割り当てられた集合の要素 a という言い方をしなくてはならない．しかし，ここでは単純に型と集合を同一視することにする．

6.3.2 モルフィズムによる意味論

ラムダ計算に対する意味関数 $\mathcal{M}[\![M]\!]_\rho$ を詳しく調べてみよう．今，ラムダ式 M 中に，n 個の自由変数 $x_1, \cdots, x_n$ が含まれているとする．M の値は，$x_1, \cdots, x_n$ に対する値に従って $\mathcal{M}[\![M]\!]_\rho$ で計算される．ここで，M の型を A, $x_1, \cdots, x_n$ の型を $C_1, \cdots, C_n$ として，$C_1 \times \cdots \times C_n$ から A への n 引数関数 f_M を

$f_M(c_1, \cdots, c_n) = \mathcal{M}[\![M]\!]_{\rho(x_1 \leftarrow c_1, \cdots, x_n \leftarrow c_n)}$

と定義する．前の $\mathcal{M}[\![M]\!]_\rho$ は環境 ρ を従えて M の意味を決めていたのに対して，f_M は M そのものの意味を表わしていると考えられる．

上の f_M は $\mathcal{M}[\![M]\!]_\rho$ を使って定義されたが，直接定義することもできる．このときに 6.2 節で議論したモルフィズムによる定義法が役に立つ．M を型 A のラムダ式，$<x_1, \cdots, x_n>$ をそれぞれ型 $C_1, \cdots, C_n$ の変数の列で変数の重複がないとする．また，M 中の自由変数はすべて変数列 $<x_1, \cdots, x_n>$ に含まれていると仮定する．このとき，$C_1 \times \cdots \times C_n$ から A への n 引数関数 $[\![M]\!]_{<x_1, \cdots, x_n>}$ を順々に定義していく．この $[\![M]\!]_{<x_1, \cdots, x_n>}$ が上の f_M と同じになる．

その前に，関数を定義するための一般的な記法を準備しておく．集合AからBへの関数fで各 $a\in A$ について $f(a)=b$ のとき，

$$f : A \to B,\ a \mapsto b$$

と書くことにする．

では，$[\![M]\!]_{<x_1,\cdots,x_n>}$ の定義を始める．

（1） Mが変数 x_i のとき，仮定から x_i は変数列$<x_1,\cdots,x_n>$の中に含まれていることに注意して，

$$[\![x_i]\!]_{<x_1,\cdots,x_n>}=\pi_i^{C_1,\cdots,C_n}$$

と定義する．ここで，$\pi_i^{C_1,\cdots,C_n}$ は直積 $C_1\times\cdots\times C_n$ の値（すなわちn項組）からi番目の要素を取り出す関数で，

$$\pi_i^{C_1,\cdots,C_n} : C_1\times\cdots\times C_n \to C_i,$$
$$<c_1,\cdots,c_n> \mapsto c_i$$

と定義する．$n=2$ の場合はすでに 6.2 節でも出てきた．

（2） Mが関数作用 M_1M_2 のとき，M_1 の型を B^A, M_2 の型を A として，

$$[\![M_1]\!]_{<x_1,\cdots,x_n>} : C_1\times\cdots\times C_n \to B^A$$
$$[\![M_2]\!]_{<x_1,\cdots,x_n>} : C_1\times\cdots\times C_n \to A$$

がすでに定義されていると仮定する．2引数関数 $App^{A,B}$ を

$$App^{A,B} : B^A\times A \to B,\ (f,a)\mapsto f(a)$$

と定義する．すなわち，$App^{A,B}$ は第1引数の関数 f を第2引数 aに作用させて $f(a)$ を求める関数を表わしている．この $App^{A,B}$ を使って，

$$[\![M_1M_2]\!]_{<x_1,\cdots,x_n>}=App^{A,B}\circ<[\![M_1]\!]_{<x_1,\cdots,x_n>},[\![M_2]\!]_{<x_1,\cdots,x_n>}>$$

と定義する．ここで，$<h_1,h_2>$は 6.2 節で述べたように，

$$<h_1,h_2> : C_1\times\cdots\times C_n \to B^A\times A,$$
$$(c_1,\cdots,c_n) \mapsto (h_1(c_1,\cdots,c_n),h_2(c_1,\cdots,c_n))$$

で定義された関数である．

（3） Mが $\lambda x.M_1$ のとき，xの型をA, M_1 の型を B として，

$$[\![M_1]\!]_{<x_1,\cdots,x_n,x>} : C_1\times\cdots\times C_n\times A \to B$$

がすでに定義されていると仮定する．変数xは M_1 中で自由変数かもしれないので，$[\![M_1]\!]_{<x_1,\cdots,x_n>}$ は定義されないことに注意されたい．また，x が変数列 $<x_1,\cdots,x_n>$ に含まれている場合には，正確にはxを別の変数名に付け変えな

くてはならない．ここで，$f=[\![M_1]\!]_{<x_1,\cdots,x_n,x>}$ として，

$$[\![\lambda x.M_1]\!]_{<x_1,\cdots,x_n>} : C_1\times\cdots\times C_n \to B^A,$$
$$(c_1,\cdots,c_n)\mapsto k$$

と定義する．ただし，k は

$$k : A\to B,\ a\mapsto f(c_1,\cdots,c_n,a)$$

と定義する．一般に，2引数以上の関数を，関数を値とする高階関数で表わすことを**カリー化** (currying) という．上の $[\![\lambda x.M_1]\!]_{<x_1,\cdots,x_n>}$ は，$f=[\![M_1]\!]_{<x_1,\cdots,x_n,x>}$ の $n+1$ 番目の引数に関してカリー化した関数を意味している．

以上のように定義した $[\![M]\!]_{<x_1,\cdots,x_n>}$ は，最初に述べた f_M とぴったり一致することが示せる．すなわち，型 $C_1,\cdots,C_n$ のすべての値 $c_1,\cdots,c_n$ について，

$$[\![M]\!]_{<x_1,\cdots,x_n>}(c_1,\cdots,c_n)=\mathcal{M}[\![M]\!]_{\rho(x_1\leftarrow c_1,\cdots,x_n\leftarrow c_n)}$$

が成り立つ．また，上の $[\![M]\!]_{<x_1,\cdots,x_n>}$ の定義は，ラムダ計算に対する 6.2 節で述べたようなモルフィズム的な表現法を説明している．上の定義では，基本関数 π_i, App と関数操作 $f\circ g, <f,g>$，カリー化だけが使われていたことに注目したい．このことは，ラムダ計算に対する新しい形式体系が定義できることを示唆している．

6.3.3 カテゴリカル・コンビネータ理論 CCL

前項までの議論をもとにして，形式体系 CCL を定義し，ラムダ計算との関連を調べる．目標は，前項で定義したラムダ計算のもう一つの意味関数である $[\![M]\!]_{<x_1,\cdots,x_n>}$ を表現できるような形式体系を作ることである．

前項の $[\![M]\!]_{<x_1,\cdots,x_n>}$ の定義では直積を使っていたので，型の定義を拡張する必要がある．6.3.1 項で述べた型の定義に 3 番目の規則として，

（3） A と B が型のとき $A\times B$ も型である．

を付け加える．2 個以上の直積 $A_1\times\cdots\times A_n$ は，2 個の直積が定義されていれば $(\cdots((A_1\times A_2)\times A_3)\times\cdots)\times A_n$ のように定義できる．

前項の $[\![M]\!]_{<x_1,\cdots,x_n>}$ は，関数を組み合わせて定義されていた．同様に，CCL の式は，すべて意味的に関数を表わし，その定義域と値域の型を明示して定義する．CCL の式を次のように定義する．ここで，CCL の式 F の型を明示するために，F の定義域の型が A，値域の型が B のとき，$F: A\to B$ と書くことに

する.

（1）　四つの関数記号 $Fst^{A,B}: A\times B\to A$, $Snd^{A,B}: A\times B\to B$, $App^{A,B}: B^A\times A\to B$, $Id_A: A\to A$

（2）　$F: A\to B$, $G: B\to C$ のとき $G\circ F: A\to C$

（3）　$F: C\to A$, $G: C\to B$ のとき $<F,G>: C\to A\times B$

（4）　$F: C\times A\to B$ のとき $\Lambda(F): C\to B^A$

上の定義で $Fst^{A,B}$ と $Snd^{A,B}$ は前項の $\pi_1{}^{A,B}$ と $\pi_2{}^{A,B}$ を意味する. $\pi_1{}^{A,B}$ と $\pi_2{}^{A,B}$ は直積 $A\times B$ の要素（すなわち2項組）のそれぞれ1番目と2番目の要素を取り出す関数なので，あえて Fst と Snd という記号を使っている. Id_A は恒等関数を表わす. 恒等関数は前項では出てこなかったが，後で CCL の公理系を定義する際に必要になる. $\Lambda(F)$ は F をカリー化した関数を表わす. すなわち，

$$\Lambda(F): C\to B^A,\quad c\mapsto k_c$$
$$k_c: A\to B,\quad a\mapsto F(c,a)$$

を意味する. 残りの App, $G\circ F$, $<F,G>$ もすべて前項と同じようにそれぞれ関数作用，関数合成，関数組を表わす.

CCL の公理は**表 6.1** のように等式の形で定義する. 表中では型に関する記述を省略してあるので，完全に型づけをしてそれぞれの等式の意味を考えてみてほしい. CCL の式の直観的な意味に戻って考えると，すべて妥当な公理であることがわかる. しかも，後で述べるように，表 6.1 の公理はラムダ計算を表現する上で十分なことが証明できる.

次に，6.3.2 項で定義した $[\![M]\!]_{<x_1,\cdots,x_n>}$ を CCL の中で形式的に表現する方法を調べる. 6.3.2 項では，$[\![M]\!]_{<x_1,\cdots,x_n>}$ は意味を考えた関数として定義され

表 6.1　CCL の公理

$(F\circ G)\circ H=F\circ(G\circ H)$
$Id\circ F=F$
$F\circ Id=F$
$Fst\circ <F,G>=F$
$Snd\circ <F,G>=G$
$<F,G>\circ H=<F\circ H,G\circ H>$
$\Lambda(F)\circ H=\Lambda(F\circ <H\circ Fst,Snd>)$
$App\circ <\Lambda(F),G>=F\circ <Id,G>$

ていたが，この項ではラムダ式から CCL の式への記号的な変換アルゴリズムを定義する．

詳しい定義を始める前に，形式化するための技術的な注意をしておく．6.3.2項では，M の型を A，$x_1,\cdots,x_n$ の型をそれぞれ $C_1,\cdots,C_n$ として，$[\![M]\!]_{<x_1,\cdots,x_n>}$ は $C_1\times\cdots\times C_n$ から A への n 引数関数と定義した．もし，M が自由変数を含まず，しかも $n=0$ の特別な場合が問題になる．この場合，$[\![M]\!]_{<>}$ は0引数関数と考えるのが自然である．問題は0引数関数の定義域の決め方である．そこで，CCL の基本型の一つとして，0引数関数の定義域を表わす特別な型1を導入する．また，$n=0$ の特別な場合も統一的に扱えるように，$[\![M]\!]_{<x_1,\cdots,x_n>}$ に対する CCL の式の型は $(\cdots((1\times C_1)\times C_2)\times\cdots)\times C_n$ とする．以後，この直積型は $1\times C_1\times C_2\times\cdots\times C_n$ と略記する．

以上を注意してラムダ式から CCL の式への変換規則を定義する．$<x_1,\cdots,x_n>$ をそれぞれ型 $C_1,\cdots,C_n$ の変数の列で，同じ変数は重複して現われないと仮定する．型 A のラムダ式 M について，M 中の自由変数がすべて変数列 $<x_1,\cdots,x_n>$ に含まれるとき，CCL の式

$$[M]_{<x_1,\cdots,x_n>}: 1\times C_1\times\cdots\times C_n\rightarrow A$$

を次のように定義する．

$$[x_i]_{<x_1,\cdots,x_n>}=Snd\circ\underbrace{Fst\circ\cdots\circ Fst}_{n-i\text{ 個}}$$

$$[M_1M_2]_{<x_1,\cdots,x_n>}=App\circ<[M_1]_{<x_1,\cdots,x_n>},[M_2]_{<x_1,\cdots,x_n>}>$$

$$[\lambda x.M_1]_{<x_1,\cdots,x_n>}=\Lambda([M_1]_{<x_1,\cdots,x_n,x>})$$

上の変換規則は単に $[\![M]\!]_{<x_1,\cdots,x_n>}$ を形式化しただけだが，形式体系としてのラムダ計算と CCL の関連を調べる上で重要である．ラムダ計算はふつう β 規則，η 規則と呼ばれる二つの公理を持つ．

（β 規則）　$(\lambda x.M)N=M[x:=N]$

ただし，$M[x:=N]$ は M 中の自由変数 x を N に置き換えた式を表わす．

（η 規則）　$\lambda x.Mx=M$

ただし，x は M 中で自由変数として現われないとする．β 規則だけを公理に持つラムダ計算を仮定すると次の定理が成り立つことが知られている．

定理．ラムダ計算で $M=N \iff$ CCL で $[M]_{<x_1,\cdots,x_n>}=[N]_{<x_1,\cdots,x_n>}$

また，ラムダ計算では式の変形を表わす**リダクション**（reduction）が定義されている．この場合，β 規則は左辺から右辺への書換え規則と考える．同様に，表 6.1 の CCL の公理を左辺から右辺への書換え規則とみなすと CCL の式のリダクションが定義される．上の定理は，等号＝をリダクションと読み換えても成り立つ．さらに，型に関する条件を取りはずして，型なしのラムダ計算と CCL を定義した場合でも同じ定理が成り立つ．

これらの定理は，ラムダ計算が CCL の中で完全にシミュレートされることを意味している．言いかえれば，CCL はラムダ計算のもう一つの表現になっているわけである．ただし，CCL は直積を操作する機構を持っているため，ラムダ計算より広い体系になっている．もし，ラムダ計算に直積を操作する機構を導入するか，もしくは CCL の直積操作を制限した体系を定義すれば，二つの体系をぴったり同じにできる．詳しくは，文献 4) を参照されたい．

β 規則と η 規則の両方を持つラムダ計算は，CCL の公理に

$$\Lambda(App) = Id$$

を付け加えた体系に対応し，上と同じ定理が成り立つ．ただし，書換え規則としての η 規則に対応する CCL の公理は複雑な形になる．

この項では，ラムダ計算の意味論に関する議論から形式体系 CCL を導入した．CCL は，ラムダ計算に対する別の形式化であると同時に，ラムダ計算の意味論を定義した代数系ともみなせる．説明を簡単にするため，型のない場合について考える．一般に，等式に基づいた形式体系 T について，集合 X と T の各式に X の要素を割り当てる関数 $[\![-]\!]$ が定義されていて，T で $M=N$ ならば X 上でも $[\![M]\!]=[\![N]\!]$ のとき，2 項組 $(X, [\![-]\!])$ は T の**モデル**（model）であるという．形式体系 T の式が $[\![-]\!]$ によって X の中で意味づけされることを意味している．今，集合 X とその要素 Id, Fst, Snd, App, および X 上の演算 $-\circ-$, $\langle -,- \rangle$, $\Lambda(-)$ が与えられていて，表 6.1 の CCL の公理を満たしていたとする．このように構造を持った集合のことを一般に**代数**（algebra）と呼ぶ．前に述べたラムダ計算と CCL の関係を示した定理によると，$(X, [-]_{\langle x_1,\cdots,x_n\rangle})$ はラムダ計算のモデルとなる．逆に，ラムダ計算のモデルはすべて CCL の公理系を満たす代数で表わせることが知られている．その意味で，CCL の公理系はラムダ計算のモデルを特徴づけた代数系と考えられる．

6.4 関数型プログラミングへの応用

ラムダ計算に **if-then-else**, リカージョンなどの機能を追加して，実際の関数型プログラム言語にまで拡張したとき，対応する CCL はどう拡張されるかを調べる．拡張 CCL は一種のプログラム言語になっているはずである．また，CCL がラムダ計算のモデルを定義していたように，拡張 CCL はもとの関数型言語の意味を記述する言語になっているはずである．

6.4.1 関数型プログラム言語 ML/O

代表的なプログラム言語である ML をモデルにして，簡単な言語 ML/O を定義する．ML/O はラムダ計算を次のように拡張した言語とする．

（1） 定数と基本関数を付け加える．

（2） リカージョンを付け加える．

（3） **if-then-else** を付け加える．

（4） 二つの値のペアを扱える．したがって，n 項組も扱える．

（5） ラムダ変数には単純な変数だけでなく，パターンを書けるようにする．

（6） 関数呼出しの際には実引数とパターンのマッチングを行なう．

ML/O の構文を表 6.2 のように決める．表の（＜式＞,＜式＞）は値のペアを表わす．式 M と N の型をそれぞれ A と B とすると，(M, N) の型は直積 $A \times B$ となる．**fun**＜パターン＞ ⇒ ＜式＞はラムダ束縛の拡張である．ラムダ計算では，2引数以上の関数はカリー化して扱っていたが，実際のプログラミングでは2引数以上の関数も直接扱ったほうが自然な場合もある．そこで，ラムダ変数の位置には単純な変数以外に変数の列も書けるようにしてある．たとえば，**fun**$(x, y) \Rightarrow M$ は2引数関数を表わし，ペア式 (N_1, N_2) に作用させることができる．また，0引数関数を表わすために **fun**$() \Rightarrow M$ のような記述も許す．0引数関数を呼び出すために＜式＞の定義中にも（）がある．6.3.4 項では，0引数関数の定義域を表わす特別な型1を導入した．型1の値は特に何であってもよいので，（）で型1の値を表わしている．型1はプログラム言語 C の void を

表 6.2　ML/O の構文規則

```
〈式〉 ::= <変数>
        | <定数>
        | <基本関数>
        | ()
        | <式> <式>
        | (<式>, <式>)
        | if<式>then<式>else<式>
        | fun<パターン>⇒<式>
        | rec<パターン>⇒<式>
        | let<パターン>=<式>in<式>
        | let rec<パターン>=<式>in<式>
〈パターン〉 ::= <変数>
        | ()
        | (<パターン>, <パターン>)
(ただし，<パターン>中の変数はすべて異なっていること)
```

考えるとわかりやすい．さらに変数の列を一般化して，

fun $((x1, x2), (y1, y2)) \Rightarrow M$

のような表現も許す．この関数を値$((1,2), (3,4))$に作用させるとパターン・マッチングが行なわれ，変数 $x1$，$x2$，$y1$，$y2$ にそれぞれ 1，2，3，4 が結合される．完全に型チェックが行なわれていれば，このマッチングは常に成功する．

関数の再帰的定義は **rec** を用いて行なう．たとえば関数 f を再帰的に定義する場合は，

rec $f \Rightarrow$ (**fun** $x \Rightarrow \cdots f \cdots$)

ないしは，

let rec $f=$(**fun** $x \Rightarrow \cdots f \cdots$) **in** $\cdots f \cdots$

のように書けばよい．**rec** の後には単純な変数だけが書ければ十分であるが，**fun** との統一性からパターンが書けるようにしてある．なお，**let rec** 式および **let** 式はそれぞれ次のような関数作用の省略形とする．

let rec $f=M$ **in** $N \Longrightarrow$ (**fun** $f \Rightarrow N$) (**rec** $f \Rightarrow M$)

let $f=M$ **in** $N \Longrightarrow$ (**fun** $f \Rightarrow N$)M

let と **let rec** は，どちらも式 N 中で使われる局所変数 f を定義するための記法である．**rec** が付くか否かは，f の定義式 M 中で f が再帰的に現われるか

否かによる．また次のような略記も許す．

$x+y \Longrightarrow +(x,y)$

$M(N) \Longrightarrow MN$

$\mathbf{let}\ f(x,y)=M\ \mathbf{in}\ N \Longrightarrow \mathbf{let}\ f=(\mathbf{fun}(x,y) \Rightarrow M)\ \mathbf{in}\ N$

6.4.2 CCL の拡張

前項でラムダ計算を拡張して言語 ML/O を定義したのに対応して，CCL の言語と公理を拡張する．もとの CCL ははじめから直積を操作する基本関数を持っていたので，ML/O で導入したペア (M,N) とパターンのラムダ束縛 **fun** $\alpha \Rightarrow M$ に対する特別な仕掛けは必要ない．定数，基本関数，**if-then-else** とリカージョンについてだけ拡張すれば十分である．

拡張 CCL の構文を表 6.3 のように定義する．新しく，*Quote*(c)，基本関数，*Cond*(F,G_1,G_2)，*Fix*(F) が付け加わっている．*Quote*(c) は値が常に c であるような定数関数を表わす．*Cond*(F,G_1,G_2) は **if-then-else** を表わす．*Fix*(F) は ML/O の **rec** 式に対応する．

表 6.3 拡張 CCL の構文規則

〈式〉 ::= *Quote*（<定数>）
　　| <基本関数>
　　| *Id* | *Fst* | *Snd* | *App*
　　| <式> ∘ <式>
　　| <<式>, <式>>
　　| *Λ*（<式>）
　　| *Fix*（<式>）
　　| *Cond*（<式>, <式>, <式>）

ML/O では，**rec** $x \Rightarrow M$ は変数 x を再帰的に定義する式であった．もう少し詳しくいうと，この **rec** 式は，

$(\mathbf{fun}\ x \Rightarrow M)v=v$

を満たす値 v を定義している．この v を **fun** $x \Rightarrow M$ の固定点（fixpoint）と呼ぶこともある．拡張 CCL の *Fix* という記号はここからきている．*Fix*(F) は，型が $C \times A \rightarrow A$ であるような F に対してだけ定義でき，*Fix*(F) 自体の型は $C \rightarrow A$ となる．ML/O の **rec** と同様に，*Fix*(F) は

$F \circ \langle Id, f \rangle = f$

を満たすような関数 f を表わす．この等式の意味は，後で述べる ML/O から拡張 CCL への変換規則から明らかになるので，ここでは説明を省略する．

もとの CCL には表 6.1 の公理が与えられていた．拡張 CCL には，新しく付け加わった式 $Quote$，$Cond(F, G_1, G_2)$，$Fix(F)$ に関連して次の四つの公理を付け加える．

$$Quote(c) \circ H = Quote(c)$$

$$Cond(F, G_1, G_2) \circ H = Cond(F \circ H, G_1 \circ H, G_2 \circ H)$$

$$Fix(F) \circ H = Fix(F \circ \langle H \circ Fst, Snd \rangle)$$

$$Fix(F) = F \circ \langle Id, Fix(F) \rangle$$

それぞれの式の意味を考えて妥当性をチェックされたい．

さて，6.3.3 項と同様に ML/O から拡張 CCL への変換規則を定義する．6.3.3 項では，ラムダ式 M と変数列 $\langle x_1, \cdots, x_n \rangle$ に対して CCL の式 $[M]_{\langle x_1, \cdots, x_n \rangle}$ を定義した．ML/O では，ラムダ変数がパターンにまで拡張されているので，変換規則でも変数列をパターンまで拡張する必要がある．ML/O の式 M とパターン γ に対して拡張 CCL の式 $[M]_\gamma$ を表 6.4 のように定義する．ただし，前と同様に M 中の自由変数はすべて γ に含まれていると仮定する．

表 6.4　ML/O から拡張 CCL への変換規則

(1)	変数 x のときさらに場合分けして
	$[x]_x = Id$
	$[x]_{(\alpha, \beta)} = \begin{cases} [x]_\alpha \circ Fst & x \text{ が } \alpha \text{ に含まれるとき} \\ [x]_\beta \circ Snd & x \text{ が } \beta \text{ に含まれるとき} \end{cases}$
(2)	定数 c のとき（特別な定数（）も含める）
	$[c]_\gamma = \mathrm{Quote}(c)$
(3)	基本関数 K のとき
	$[K]_\gamma = \Lambda(K \circ Snd)$
(4)	$[(M, N)]_\gamma = \langle [M]_\gamma, [N]_\gamma \rangle$
(5)	$[MN]_\gamma = App \circ \langle [M]_\gamma, [N]_\gamma \rangle$
(6)	$[\mathbf{fun}\ \alpha \Rightarrow M]_\gamma = \Lambda([M]_{(\gamma, \alpha)})$
(7)	$[\mathbf{rec}\ \alpha \Rightarrow M]_\gamma = Fix([M]_{(\gamma, \alpha)})$
(8)	$[\mathbf{if}\ M\ \mathbf{then}\ N_1\ \mathbf{else}\ N_2]_\gamma = Cond([M]_\gamma, [N_1]_\gamma, [N_2]_\gamma)$

表 6.4 の基本関数 k の変換は奇妙な形をしているが，次のように考えると自然であることがわかる．ML/O で基本関数 K を式 M に作用させた式 KM を拡張 CCL に変換すると

$$[KM]_\Gamma = App \circ \langle \Lambda(K \circ Snd), [M]_\Gamma \rangle$$

となる．さらに CCL の公理を使って右辺を整理すると

$$[KM]_\Gamma = K \circ [M]_\Gamma$$

となる．ML/O でも K が関数であることを考えると，最後の等式は自然である．

ML/O から拡張 CCL への変換規則を見ると，先ほど説明を省略した $Fix(F)$ の意味がはっきりする．ML/O での **rec** の意味を考えると

$$(\mathbf{rec}\ \alpha \Rightarrow M) = (\mathbf{fun}\ \alpha \Rightarrow M)\ (\mathbf{rec}\ \alpha \Rightarrow M)$$

の等式が成り立つ．両辺に変換規則を適用すると，拡張 CCL の等式

$$Fix([M]_{(\Gamma,\alpha)} = App \circ \langle \Lambda([M]_{(\Gamma,\alpha)}), Fix([M]_{(\Gamma,\alpha)}) \rangle$$

が得られる．さらに右辺を App に関する CCL の公理を使って書き直すと

$$Fix([M]_{(\Gamma,\alpha)}) = [M]_{(\Gamma,\alpha)} \circ \langle Id, Fix([M]_{(\Gamma,\alpha)}) \rangle$$

となる．この式が正に $Fix(F)$ の意味を表わしている．

もとの言語 ML/O の式と拡張 CCL へ変換した後の式の違いは，関数型言語の処理系を設計する立場から考えるとわかりやすい．ML/O のプログラムの各部分式を実行するためには，その部分式に自由変数が含まれている場合は，自由変数の現在値を参照する操作が必要である．6.3.1 項で述べたように，自由変数とその値の対応表を**環境**と呼ぶ．ML/O の処理系では環境を操作する機構が必要である．一方，拡張 CCL の式には変数が存在しない．各部分式は単独で関数を表わしている．したがって，式の外にある環境などの情報を参照する必要がない．しかし，環境の概念は CCL の中でも形を変えて存在している．ML/O の式 M 中に自由変数 $x_1, \cdots, x_n$ が含まれていたとすると，拡張 CCL へ変換した後の式 $[M]_{\langle x_1,\cdots,x_n\rangle}$（正確には変数列$\langle x_1, \cdots, x_n \rangle$ではなくパターンとしなければならない）は，$x_1, \cdots, x_n$ に対する値をもらって M の値を返す関数を意味する．言いかえれば，$[M]_{\langle x_1,\cdots,x_n\rangle}$ は環境を引数とする関数になっている．事情は 6.3.2 項で述べたことと同じである．拡張 CCL では，環境への参照の機構が自然に言語の中に取り込まれていると考えられる．したがって，拡張 CCL の処理系では環境操作の機構を特別に作る必要がない．事実，拡張 CCL の式はほとんどそのままで機械語コードとみなせる．たとえば，基本関数はサブルーティン呼出しだし，関数合成 $F \circ G$ は G を実行し次に F を実行

する逐次処理とみなせる.

ML/O から拡張 CCL への変換の例を挙げておく. ML/O のプログラム

$$\textbf{let } f(x) = (\textbf{let } g(y) = x - y \textbf{ in } x + g(2 * x)) \textbf{ in } f(3)$$

を拡張 CCL へ変換すると,

$$\begin{aligned}
&[(\textbf{fun } f \Rightarrow f(3)) \\
&\quad (\textbf{fun } x \Rightarrow (\textbf{fun } g \Rightarrow +(x, g(*(2, x)))) \\
&\qquad\qquad (\textbf{fun } y \Rightarrow -(x, y)))]_r \\
&= App \circ < Acc(0),\ Quote(3) > \\
&\qquad \circ < Id, \Lambda(+ \circ < Acc(1), \\
&\qquad\qquad\qquad App \circ < Acc(0),\ * \circ < Quote(2), Acc(1) >>> \\
&\qquad\qquad \circ < Id, \Lambda(- \circ < Acc(1), Acc(0) >) >) >
\end{aligned}$$

が得られる. ただし, 各 n について

$$Acc(n) = Snd \circ \underbrace{Fst \circ \cdots \circ Fst}_{n\,個}$$

と定義する. また, 変換後の式を App に関する公理を使って整理してある. この拡張 CCL の式の $Acc(n)$ の部分が, もとの ML/O のプログラムで変数を参照していたところに対応する.

6.4.3 CCL に基づいた処理系

前節の議論をもとに, 抽象的な機械を設計して, 拡張 CCL の式をその機械語に変換することを考える. つまり, ML/O のプログラムを拡張 CCL を経由して機械語コードに落とすコンパイラが出き上がるわけである.

今まで, 式の計算方法に関する議論をとばしてきた. ML/O の式も CCL の式も単に関数を表わしている記述として扱ってきた. その関数の値を計算するための規則が必要である. ここでは, ML, Lisp などと同じように作用順序 (applicative order), すなわち一番内側の部分式から順に評価する計算規則を採用する.

以下, 順を追って抽象機械を設計していく. 拡張 CCL の各式は入力をして出力をする関数の形をしているので, 入力と出力の値を入れるレジスタを用意する. 拡張 CCL の式に対する機械語コードは, 常にレジスタの値を入力して

レジスタへ計算結果を出力するという形に設計する．また，スタックを一つ用意する．このスタックは CCL の関数組<*F*, *G*>を処理するときに必要になる．関数組 ⟨*F*, *G*⟩ を実行するためには次のような処理をすればよい．（1）*F*を実行，（2）*G*を実行，（3）それぞれの実行結果のペアを作る．したがって，*G*の実行まで入力を保存しておかなければならないし，（3）のペアを作るまで（1）の*F*の実行結果を保存しておく必要もある．そのためにスタックを使う．

次にデータの実現方法を決めなくてはならない．ML/O のデータ型は基本型，直積型，関数型の3種類であった．基本型の実現方法は個別に扱うことにしてここでは考えない．直積型のデータであるペアは二分木で表わすことにする．二分木を操作する機械語命令 *car*, *cdr*, *cons* を用意する．関数型のデータは，その関数を実行する機械語コードの入り口番地と環境の対で表わす．一般に，関数と環境の対は**関数閉包** (function closure) と呼ばれている．関数型のデータは，$[\mathbf{fun}\ \alpha \Rightarrow M]_{\Gamma} = \Lambda([M]_{(\Gamma,\alpha)})$ で作られる．この式を実行すると，*M* 中の自由変動の現在値と *M* の実行のための機械語コードの入り口番地を対にした関数閉包が作られるように設計する．

以上の設計方針に基づいて，機械語命令と拡張 CCL の式から機械語コードへの変換規則をそれぞれ**表 6.5** と**表 6.6** のように定義する．表 6.6 では，拡張

表 6.5 機械語命令

実行前			実行後		
レジスタ	スタック	コード	レジスタ	スタック	コード
(a, b)	S	*car* ; C	a	S	C
(a, b)	S	*cdr* ; C	b	S	C
a	S	*push* ; C	a	$S.a$	C
a	$S.b$	*cons* ; C	(a, b)	S	C
a	$S.b$	*swap* ; C	b	$S.a$	C
a	S	*load*(c) ; C	c	S	C
a	S	*call*(K) ; C	a	$S.C$	K
true	$S.a$	*branch*$(C1, C2)$; C	a	S	$C1$
false	$S.a$	*branch*$(C1, C2)$; C	a	S	$C2$
a	S	*closure*(P) ; C	$(P : a)$	S	C
$(P : a, b)$	S	*app* ; C	(a, b)	$S.C$	P
a	$S.C$	*return*	a	S	C
a	S	*noop* ; C	a	S	C

表 6.6　拡張 CCL から機械語への変換規則

$[\![F \circ G]\!]=[\![G]\!]\,;\,[\![F]\!]$
$[\![<F, G>]\!]=push\,;\,[\![F]\!]\,;\,swap\,;\,[\![G]\!]\,;\,cons$
$[\![Fst]\!]=car$
$[\![Snd]\!]=cdr$
$[\![Id]\!]=noop$
$[\![Quote\,(c)]\!]=load(c)$
$[\![K]\!]=call(K)$　　(Kは基本関数)
$[\![Cond(F, G_1, G_2)]\!]=push\,;\,[\![F]\!]\,;\,branch([\![G_1]\!], [\![G_2]\!])$
$[\![\Lambda(F)]\!]=closure(P)$　　$(P : [\![F]\!]\,;\,return\,;)$
$[\![App]\!]=app$
$[\![Fix(F)]\!]=fix\,1\,;\,[\![F]\!]\,;\,fix\,2$

CCL の式 F に対する機械語の命令列が $[\![F]\!]$ で表わされている．セミコロン「;」は機械語命令の区切り記号を表わす．リカージョンに対応する Fix を処理する命令 $fix\,1$ と $fix\,2$ は，後で説明することにして表 6.5 には載せていない．以下，$\Lambda(F)$，App，$Fix(F)$ についてだけ説明する．ほかの式については，表 6.5 と表 6.6 の説明で十分と思われる．

ML/O の **fun** 式は関数型なので，$[\![\mathbf{fun}\,\alpha \Rightarrow M]\!]_\gamma=\Lambda([\![M]\!]_{(\gamma,\alpha)})$ を実行した結果は，γ に関する環境と M を評価するための機械語コードの入り口番地を対にした関数閉包になる．表 6.6 の $[\![\Lambda(F)]\!]$ の定義では，$[\![F]\!]$ のコードをサブルーティンとして，その入り口ラベルを P とすることを表わしている．表 6.5 の命令 $closure(P)$ の定義は，サブルーティン P とレジスタの値 a を対にした関数閉包 $(P : a)$ を作ることを意味している．関数閉包の実際の実行は，命令 $[\![App]\!]=app$ で行なわれる．前に述べたように，$App : B^A \times A \rightarrow B$ は，第1引数の関数を第2引数に作用させる関数を表わしていた．表 6.5 の命令 app の定義は，関数閉包と値のペア $(P : a, b)$ を受け取り，ペア (a, b) を作ってサブルーティン P を呼び出すことを意味している．サブルーティン P の実行後は app の次の命令に戻らなくてはならないので，戻り番地がスタックに入れられている．また，サブルーティンのコードの最後には，表 6.6 の $[\![\Lambda(F)]\!]$ の定義のように命令 $return$ を付け加える．

リカージョンに対応する $Fix(F)$ の処理は複雑である．6.4.2 項で説明したように，$\mathbf{rec}\;x \Rightarrow M$ は $\mathbf{fun}\;x \Rightarrow M$ の fixpoint，すなわち

$$(\mathbf{fun}\;x \Rightarrow M)v=v$$

を満たすような値vを表わす．したがって，最初xの値を **undef** として式Mを評価して，後で計算結果の中に含まれている **undef** を計算結果自身で置き換えればよい．しかし，Mの実行中でまだ値が決まっていない x を参照するのは奇妙である．リカージョン **rec** の厳密な意味の決め方にもよるが，もしMの評価中に x を参照した場合はエラーとしてもよいし，あるいはリスト構造を扱っているような場合には，x の値は **undef** のままで実行を続けてもよい．ただし，**rec** 文が関数の再帰的定義に使われている場合には，Mの評価中でxは参照されないので問題は起こらない．たとえば，**rec** $x \Rightarrow$ (**fun** $y \Rightarrow \cdots x \cdots$) のような場合，**fun** 以下の式は単に関数閉包を作るだけなので，x の値が参照されることはない．

以上に注意すると $[\mathbf{rec}\ x \Rightarrow M]_r = Fix([M]_{(r,x)})$ の実行は，次の手順を踏めばよいことがわかる．

（1） レジスタに入れられている環境 a と特別な値 **undef** を *cons* して新しい環境 (a, **undef**) を作りレジスタに戻す．

（2） あとで **undef** を実際の値に置き換えるために，(1)で作った新しい環境をスタックにも入れる．

（3） $[M]_{(r,x)}$ に対する命令列を実行する．

（4） (2)で保存したもとの環境をスタックから取り出し，**undef** の部分を(3)の実行結果で置き換える．

図 6.1 に以上の操作を図示してある．(1)と(2)の操作を命令 $fix1$ で表わし，(4)を $fix2$ で表わしている．図 6.1 の最後で，セルがごみになっているように見えるが，一般には値bの中からポインタで参照されている．たとえば，リカージョンの典型的な場合である次のプログラムを実行すると，**図 6.2** のようなデータが作られる．

$$[\mathbf{rec}\ f \Rightarrow (\mathbf{fun}\ z \Rightarrow M)]_r = Fix(\Lambda([M]_{((r,f),z)}))$$

図 6.2 で，a は実行前のレジスタの値を表わし，P は $[M]_{((r,f),z)}$ を実行するサブルーティンの入り口番地を表わしている．この関数閉包を値に作用させたときの動きを調べると，再帰呼出しが実現されていることが確かめられる．図 6.2 のようなループ構造が再帰呼出しのキーになっている．

上の処理手順は **rec** の次が単純な変数の場合であるが，一般には **rec** の次に

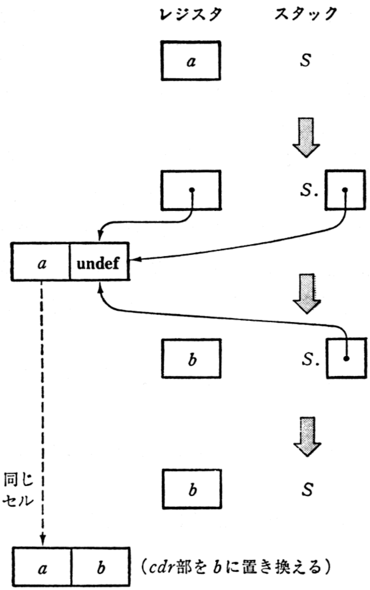


図 6.1　$Fix(F)$の処理

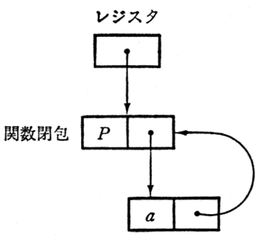


図 6.2　**rec** $f \Rightarrow$ (**fun** $z \Rightarrow M$) の実行結果

はパターンが書けると定義した．一般の場合は，(1)の新しい環境を作る際にパターンに対応したポインタ構造を作る必要がある．

6.4.4　公理系に基づいた最適化

ML/O のプログラムは，拡張 CCL を経由して機械語まで落とせばとにかく実行できる．しかし，このままではむだな部分が多過ぎる．現実的なコンパイラにするためには最適化が必要である．関数型プログラムの最適化の手法はいろいろ考えられているが，そのうちのいくつかは拡張 CCL の公理系の中で統一的に説明できる．拡張 CCL の式は，公理系に基づいて変形をしても，意味的に元の式と同じ関数を表わすことが保障される．公理系に基づいて式を簡単化してから機械語に落とせば最適化が実現できるはずである．ただし，拡張 CCL の公理系は純粋に関数としての同値性しか保証していないことに注意しな

くてはならない．効率の問題は，拡張 CCL の式の計算手順を固定したときでないと議論できない．公理系そのものは，二つの式でどちらが効率が良いかを決める機能を持たない．したがって，式をどう変形したらよいかは個々に考えなくてはならない．以下，効率化のためのいくつかの具体例を紹介する．

6.4.3 項で述べた ML/O の処理系の一番の問題は，変数を参照するのに時間がかかることである．変数の現在値を表わす環境が二分木で表現されていたので，変数を参照するためには，*car* と *cdr* を使って二分木の中の対応する場所の値を取り出さなくてはならない．変数の参照時間を短くするためには，環境を表わす二分木をなるべく小さくする必要がある．

環境を小さくするための典型的な方法を述べる．たとえば，ML/O の式 M 中の自由変数が x ただ一つだけだったとすると，拡張 CCL の式 $[M]_\gamma$ は

$$[M]_\gamma = [M]_x \circ [x]_\gamma$$

と変形できる．すなわち，与えられた環境の中の x の値だけを M の処理にわたすコードに変形できる．一般に，α, β, γ をパターン，z を β 中の変数で，$\gamma = \beta[z := \alpha]$ とする．すなわち，α が γ の一部であることをいっている．M 中の自由変数がすべて α に含まれるときは，

$$[M]_\gamma = [M]_\alpha \circ [z]_\beta$$

と変形でき，M の実行中の環境を小さくできる．しかも，この等式は拡張 CCL の公理系の中で形式的に証明される．

もう少し複雑な例をあげる．局所関数を使った次のプログラムの実行を考える．

let $f = ($**fun** $\alpha \Rightarrow M)$ **in** N

局所関数 f が定義されている．この式を拡張 CCL に変換すると

$$App \circ <\Lambda([N]_{(\gamma,f)}), \Lambda([M]_{(\gamma,\alpha)})>$$

となり，さらに App に関する公理を使って

$$[N]_{(\gamma,f)} \circ <Id, \Lambda([M]_{(\gamma,\alpha)})>$$

と簡単化される．この式を見ると，局所関数 f が関数閉包の形で環境の一部として N の実行にわたっていることがわかる．しかし，f に対する機械語コードはコンパイル時にわかるし，M 中の自由変数の値も N の実行中にわかるはずなので，f に対する関数閉包を環境の一部に含ませる必要はない．最初

から M を実行するサブルーティンを作っておき，N 中で f が呼ばれたらそのサブルーティンを呼び出せばよい．

この実現法は，拡張 CCL に基づいて統一的に説明できる．まず，新しい関数記号 K を導入して

$$K=[M]_{(\gamma,\alpha)}$$

と定義し，次のコードを作る．

$$K : [\![[M]_{(\gamma,\alpha)}]\!]\,;\ \boldsymbol{return}\,;$$

新しく導入した K を基本関数とみなして，その処理サブルーティンを定義したと考える．次に，式 N 中の f を，

$$\mathbf{fun}\ x \Rightarrow K(\gamma, x)$$

と置き換えた式を N' として，**let** 式に対するコードは $[\![[N']_\gamma]\!]$ を生成する．この処理が正しいことは，拡張 CCL の公理系を使って形式的に示せる．すなわち等式 $K=[M]_{(\gamma,\alpha)}$ を仮定すると，

$$[N']_\gamma=[\mathbf{let}\ f=(\mathbf{fun}\ \alpha \Rightarrow M)\ \mathbf{in}\ N]_\gamma$$

が拡張 CCL の公理系の中で証明できる．

もう少し詳しく調べて，インプリメント的にも正しいことを確かめてみる．式 N 中で f が関数作用 fL の形で現われた場合と単独に現われた場合に分けて考える．最初の場合，fL の部分は $[N']_\gamma$ の中では $[(\mathbf{fun}\ x \Rightarrow K(\gamma, x))L]_\beta$ となっている．ML/O から拡張 CCL への変換規則によると，β は

$$\beta=(\cdots((\gamma,\beta_1),\beta_2),\cdots,\beta_n)$$

の形をしていることに注意されたい．さらに公理を使って簡単化すると，

$$[(\mathbf{fun}\ x \Rightarrow K(\gamma, x))L]_\beta=K\circ\langle[\gamma]_\beta, [L]_\beta\rangle$$

となる．この式は，$\mathbf{fun}\ \alpha \Rightarrow M$ が定義された時点の環境と L を実行した結果とのペアを作り，サブルーティン K を呼び出すことを意味している．同様に，式 N 中で f が単独に現われた場合は，

$$[\mathbf{fun}\ x \Rightarrow K(\gamma, x)]_\beta=\Lambda(K)\circ[\gamma]_\beta$$

となる．この式は，$\mathbf{fun}\ \alpha \Rightarrow M$ が定義された時点の環境とサブルーティン K との対からなる関数閉包を作ることを意味している．

実際にコードを生成する際には，$[N']_\gamma$ そのものではなく上のように簡単化してからコードに変換する必要がある．また，$[\gamma]_\beta$ を正直にコードに変換する

と，まず与えられた環境から γ 中の各変数の値を取り出してから，新たに γ のパターンに対応する環境コードを作ることになってしまう．しかし，$\beta=(\cdots((\gamma,\beta_1),\beta_2),\cdots,\beta_n)$ であることに注意すると，$[\gamma]_\beta$ の代わりに

$$\underbrace{Fst\circ\cdots\circ Fst}_{n\text{ 個}}$$

としてもよいことがわかる．

局所関数に関する簡単化は **rec** 式にも適用できる．関数の再帰的定義は

$$[\mathbf{rec}\ f\Rightarrow(\mathbf{fun}\ \alpha\Rightarrow M)]_\gamma=Fix(\Lambda([M]_{((\gamma,f),\alpha)}))$$

と変換される．前と同じように，新しい関数記号 K を導入して，サブルーティン

$$K:[\![[M']_{(\gamma,\alpha)}]\!];\ return;$$

を作る．ただし，M' は M 中の f を

$$\mathbf{fun}\ x\Rightarrow K(\gamma,x)$$

に置き換えた式を表わす．**rec** 式自体は $\Lambda(K)$ に変換して，$closure(K)$ のコードを生成する．このインプリメントが正しいことは，前と同様に形式的に証明できる．等式

$$\Lambda(K)=[\mathbf{rec}\ f\Rightarrow(\mathbf{fun}\ \alpha\Rightarrow M)]_\gamma$$

を仮定すると，

$$\Lambda(K)=\Lambda([M']_{(\gamma,\alpha)})$$

が拡張 CCL の公理系の中で証明できる．すなわち，K を上のようなサブルーティンで実現してもよいことが証明されるわけである．

拡張 CCL の公理系から導ける等式をいくつか紹介したが，これらはすべて次の定理から簡単に導ける．

定理 (i) $[M]_\alpha\circ[\alpha[z:=N]]_\beta=[M[z:=N]]_\beta$

(ii) $[M]_{\alpha[z:=\gamma]}\circ[\alpha]_\beta=[M]_{\beta[z:=\gamma]}$

下にまとめた効率化のための変形が正しいことを，上の定理を使って証明するのもよい練習問題である．

(1) $[M]_{\beta[z:=\alpha]}=[M]_\alpha\circ[z]_\beta$

(2) $K=[M]_{(\gamma,\alpha)}\Longrightarrow[N[f:=(\mathbf{fun}\ x\Rightarrow K(\gamma,x))]]_\gamma=[\mathbf{let}\ f=(\mathbf{fun}\ \alpha\Rightarrow M)\ \mathbf{in}\ N]_\gamma$

(3) $\Lambda(K)=[\mathbf{rec}\ f\Rightarrow(\mathbf{fun}\ \alpha\Rightarrow M)]_\gamma$

$$\Longrightarrow \Lambda(K)=\Lambda([M[f:=(\mathbf{fun}\ x \Rightarrow K(\gamma,x))]]_{(\gamma,\alpha)})$$

この章の目的は，関数型言語の現実的な処理系を作ることではないが，一つだけ注意を付け加えておく．環境は二分木で表わされていたが，常に

$$(\cdots((( ),a_1),a_2),\cdots,a_n)$$

の形で表われることに注意したい．すなわち，n 項組として表われるのである．したがって，環境はメモリの連続領域に置くことができるし，またそのほうが効率良く変数の値を取り出せる．詳しくいうと，環境は **rec** の外では固定領域に，**rec** の内ではスタックに，関数閉包内ではヒープの連続領域に入れることができる．環境を二分木で表わしたのは単に便宜上のことである．本来なら，環境を表わすデータ型はペア型とは別にすべきである．このことは，ラムダ計算と CCL の関係についてもいえる．CCL の直積に関する基本操作は，環境を操作するためと直積型データそのものを操作するための2通りに使われる．そのため，6.3.3 項で述べたように CCL はラムダ計算より広い体系になっている．環境操作のためだけに直積が使われるように CCL の定義を修正すると，ラムダ計算とぴったり同じ体系ができる．

6.5　ま　と　め

カテゴリ理論のスローガンの一つ「矢印ですべてを表わそう」をもとにして，ラムダ計算とその拡張である関数型言語をモルフィズムで表わすとどうなるかを述べてきた．この章の表題は「モルフィズム・プログラミング」としたほうが良かったかもしれない．でき上がった言語（CCL）では，すべての構成要素が関数（モルフィズム）で表わされていた．その意味で，真の関数型言語といえるかもしれないが，このままの形で，実際のプログラム言語として使えるとは思えない．しかし，もし第5章で紹介した FP がプログラミングに有効であれば，CCL 単独かあるいはラムダ式を組み合わせて新しいプログラム言語が作れる可能性もある．ここでは述べなかったが，拡張 CCL から ML/O への変換規則も定義できる．したがって，ML/O と拡張 CCL を混在させた言語も定義できる．この章で述べたモルフィズムによる表現法は，ラムダ計算，関数型言語に限らずいろいろなところで使える．たとえば，論理式ないし論理型

プログラム言語に応用したらどうなるかを調べるのも興味がある．事実，文献 14）ではカテゴリに基づいた単一化（unification）アルゴリズムが紹介されている．

この章で扱った CCL は，カテゴリの分野で**カルテジアン閉カテゴリ** CCC（cartesian closed category）と呼ばれる理論とラムダ計算のモデルとの関連を調べた研究から生まれた形式体系である．ラムダ計算と CCC との関連については文献 1）と 2）などの本がある．CCL については文献 3）を参照されたい．カテゴリカル・コンビネータという言葉は文献 3）の著者の命名である．CCL とラムダ計算の関連については文献 4）でも詳しく述べられている．6.4 節で述べた抽象機械は CAM（categorical abstract machine）と呼ばれ，文献 5）で解説されている．この著者のグループでは，ML の一つの方言になる CAML と呼ぶ関数型言語を設計し，CAM に基づいて実際のコンパイラを開発している．ML については，文献 6）などが良い教科書である．ただし，6）は ML の創始者のグループによって提案された Standard ML に基づいて書かれている．関数型言語の処理系についてはいろいろ考案されているが，たとえば文献 7）で解説してある手法と CCL に基づいた処理法とを比較してみるのも興味深い．カテゴリカル・コンビネータを関数型言語の処理系に応用した例は文献 8）と 9）でも述べられている．

本章で扱ったカテゴリ理論の応用例では，プログラミングそのものにカテゴリを利用したわけではないが，カテゴリ理論を直接プログラミングに応用した例が文献 10）に見られる．文献 10）では，カテゴリ理論で用いられている手法を使って，データ型が統一的に定義できるプログラム言語が提案されている．今後の発展が期待される分野である．

本章では，カテゴリの定義をぬきに解説してきた．カテゴリ理論の教科書である文献 11）から 14）などを参照されたい．特に，12）は計算機科学者向けに書かれている．14）は計算機科学の立場から見たカテゴリ理論について書かれている．この本では，カテゴリ理論の諸概念を関数型言語 ML のプログラムで表現する試みがなされている．今後カテゴリがプログラム言語にどう応用されていくかを想像する上でも参考になる．カテゴリ理論を計算機科学に応用した研究については，文献 15）から 19）の会議録などが参考になる．また，計

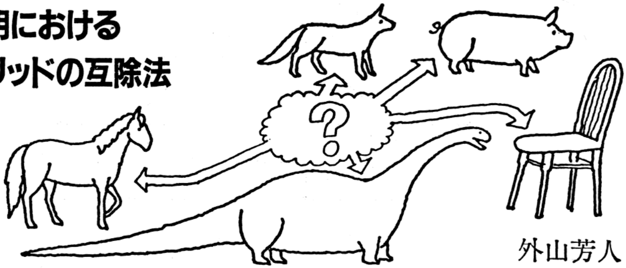
算機科学における カテゴリ あるいは 代数 に関する研究論文を専門に扱う雑誌 20) が 1991 年から刊行される予定である.

参考文献

1) J. Lambek and P. J. Scott : *Introduction to Higher Order Categorical Logic*, Cambridge University Press, 1986.
2) C. P. J. Koymans : *Models of the Lambda Calculus*, CWI, 1984.
3) P.-L. Curien : *Categorical Combinators, Sequential Algorithms and Functional Programming*, Pitman, 1986.
4) H. Yokouchi and T. Hikita : A rewriting system on categorical combinators with multiple arguments, *SIAM Journal on computing*, Vol. 19, No. 1, pp. 78–97, 1990.
5) G. Cousineau, P.-L. Curien, and M. Mauny : The categorical abstract machine, *LNCS 201*, Springer-Verlag, pp. 50–64, 1985.
6) A. P. Wikströn : *Functional Programming Using Standard ML*, Prentice Hall, 1987.
7) S. L. Peyton Jones : *The Implementation of Functional Programming Languages*, Prentice Hall, 1987.
8) 井田哲雄 : ラムダ計算とそのモデル「カルテシアン閉カテゴリ」による COMMON LISP の解釈と新しい処理系, コンピュータ・ソフトウェア, Vol. 4, No. 4, pp. 33–44, 1987.
9) R. D. Lins : On the efficiency of categorical combinators as a rewriting system, *Software——Practice & Exprerience*, Vol. 17, No. 8, pp. 547–559, 1987.
10) 萩野達也 : カテゴリー理論的関数型プログラミング言語, コンピュータ・ソフトウェア, Vol. 7, No. 1, pp. 16–32, 1990.
11) S. MacLane : *Categories for the Working Mathematician*, Springer-Verlag, 1971.
12) M. A. Arbib and E. G. Manes : *Arrows, Structures, and Functors——The Categorical Imperative*, Academic Press, 1975.
13) R. Goldblatt : *Topoi——The Categorical Analysis of Logic*, North-Holland, 1971.
14) D. E. Rydeheard and R. M. Burstall : *Computational Category Theory*, Prentice Hall, 1988.
15) D. Pitt, S. Abramsky, A. Poigné and D. Rydeheard (Eds.) : *Category Theory and Computer Programming*, LNCS 240, Springer-Verlag, 1986.
16) D. H. Pitt, A. Poigné and D. E. Rydeheard (Eds.) : *Category Theory and Computer Science*, LNCS 283, Springer-Verlag, 1987.
17) D. H. Pitt, D. E. Rydeheard, P. Dybjer, A. M. Pitts and A. Poigné (Eds.) : *Category Theory and Computer Science*, LNCS 389, Springer-Verlag, 1989.
18) H. Ehrig, H. Herrlich, H.-J. Kreowski and G. Preuß (Eds.) : *Categorical Methods in Computer Science*, LNCS 393, Springer-Verlag, 1989.
19) AMS : *Categories in Computer Science and Logic*, Contemporary Mathematics, AMS, 1988.
20) *Mathematical Structures in Computer Science*.

（よこうち ひろふみ）

第7章 最大公約数——普遍代数,多項式イデアル,自動証明におけるユークリッドの互除法



外山芳人

7.1 はじめに

新しいプログラミング・パラダイムに関連したさまざまな話題が取り上げられてきたが,本章では**パラダイムの最大公約数**について話そうと思う.最大公約数から読者のみなさんがすぐに思い付くのは**ユークリッドの互除法**であろう.この最大公約数を求めるための有名なアルゴリズムが,実はそれ自身いくつかのパラダイムの中心となる最大公約数であったという最近の驚くべき発見が本章のテーマである.まず話の発端はコンピュータ・サイエンスが誕生して間もない1960年代にさかのぼる.

今から4半世紀前,1965年前後にコンピュータ・サイエンスの三分野で独立に,三つの重要なアルゴリズムが発見された.**普遍代数**の語問題を解くための**Knuth-Bendix**(クヌース・ベンディクス)**完備化アルゴリズム**,**多項式イデアル**の標準基底を求める**Buchberger**(ブフバーガ)**アルゴリズム**,そして一階述語論理の**定理自動証明法**である**レゾリューション法**である(図 7.1).その後,三つのアルゴリズムは互いに関係し合うことなく,それぞれの分野で,独立に発展してきた.そして,Knuth-Bendix 完備化アルゴリズムは代数的仕様記述や関数型プログラミング言語などの等式システムの分野で,Buchberger アルゴリズムは数式処理の分野で,レゾリューション法は定理自動証明や論理型プログラミング言語の分野で,それぞれ中心的な役割を果たすに至ったことは広く知られているとおりである.

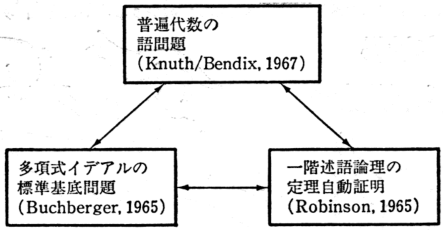


図 7.1　最大公約数は何だろう？

この三つのアルゴリズムは，目的も，適用分野も，外見も，そして進化の歴史も，まったく異なっている．ところが驚くべきことに，これらのアルゴリズムはすべて同じ原理に基づいていることが最近明らかになった．三つのアルゴリズムは，人類が発見した最古のアルゴリズムの一つ，あのよく知られている**ユークリッドの互除法**の進化した子孫たちだったのである．

これらのアルゴリズムを統一的に結び付ける鍵，つまり三つのアルゴリズムの最大公約数となるのは，**完備化アルゴリズム**というアイデアである．この最大公約数によって，三つの世界がどのように結び付いていたのかが一望のもとに見渡せる（図 7.2）．Buchberger アルゴリズム，Knuth-Bendix 完備化アルゴリズム，レゾリューション法は，それぞれ多項式環，普遍代数，一階述語論理における完備化アルゴリズムであり，ユークリッドの互除法はユークリッド環

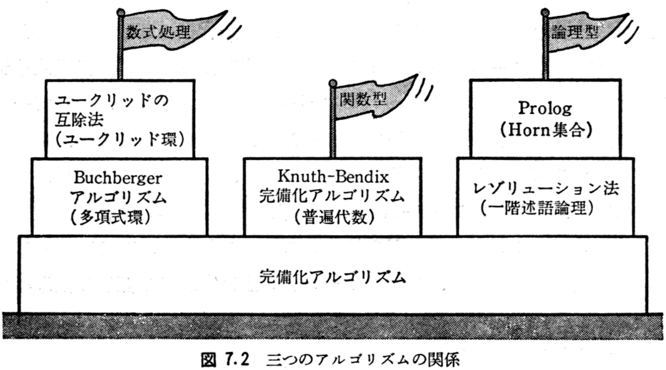


図 7.2　三つのアルゴリズムの関係

における完備化アルゴリズム，Prolog で用いられる SLD レゾリューション法は Horn（ホーン）集合における完備化アルゴリズムになる．

このように，完備化アルゴリズムという統一的な視点のもとで全体をながめると，これまでばらばらに研究されてきた三つの世界の思いがけない結付きが浮かび上がってくるとともに，4 半世紀にわたって独自に蓄積されてきたそれぞれの研究成果の補完関係も容易に読み取ることができる．以下では，この非常に魅力的なパラダイムのエッセンスをできるだけ簡単に紹介してみよう．

7.2 完備化アルゴリズム

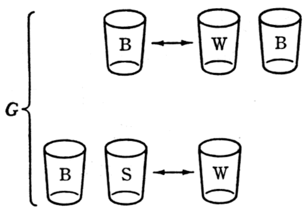
まず以下のパズルを用いて完備化アルゴリズムを説明しよう．

7.2.1 グラス置換えパズル

ウイスキーのグラス W，ビールのグラス B，酒のグラス S が一列に並んでいる．



このとき，次のグラス置換え規則 G によって，グラス列を置き換えることが許されるものとする．



各グラスを以下では W，B，S で表わすことにすると，置換え規則 B↔WB は，B を 2 個のグラスの並び WB に置き換えてもよいし，その逆に WB を 1 個のグラス B に置き換えてもよい，ということを示している．たとえば，最初に示したグラス列 α=WBS は以下のようにしてグラス列 β=BSBS に置き換えるこ

とができる．

$$\underbrace{\mathrm{WBS}}_{\alpha} \Longleftrightarrow \mathrm{WWBS} \Longleftrightarrow \mathrm{BSWBS} \Longleftrightarrow \underbrace{\mathrm{BSBS}}_{\beta}$$

このときαとβは置換え規則Gを用いてつなぐことができるという．グラス列αとβが0回以上の置換えによってつながるなら$\alpha \overset{*}{\Longleftrightarrow} \beta$と書く．それでは以下のグラス列はうまくつながるだろうか．

問題1　(1)　$\mathrm{BSBS} \overset{*}{\Longleftrightarrow} \mathrm{WWWB}$?

　　　　(2)　$\mathrm{BSBBS} \overset{*}{\Longleftrightarrow} \mathrm{BWW}$?

この問題の難しさは，二つのグラス列αとβをつなぐことが不可能な場合に，どうやって不可能であることを示すかという点にある．αとβをつなぐことが可能ならば，試行錯誤を繰り返して一つだけ答を発見すればよい．しかし，不可能の場合には，すべての置換えがαとβをつなぐことに失敗することを示す必要がある．置換え方法は無限通りあるから，これらを一つ一つ検査していては永久に答は得られない．

このような問題をうまく解く手法の一つはリダクションシステムを用いることである．実際，グラス置換えパズルは完備なリダクションシステムを用いることによって非常に簡単に解くことができる．以下ではリダクションシステムによるパズルの解法について説明しよう．

7.2.2　リダクションシステム

グラス置換え規則Gは両方向のグラスの置換えを許している．たとえば，置換え規則 B↔WB を用いて，B を WB に置き換えることも，逆に WB を B に置き換えることも可能である．したがって，グラス列αから何回かの置換えによってグラス列βが得られたならば，置換えを逆にたどることによってβからαを得ることもできる．

このような両方向の置換え規則 B↔WB を一方向の置換え規則に制限してみよう．可能性としては，B を WB に置き換える規則 B→WB と，WB を B に置き換える規則 WB→B が考えられる．より簡単なグラス列へ置換えが進んでいくように，ここでは WB→B を選ぶ．同様に置換え規則 BS↔W を BS→W

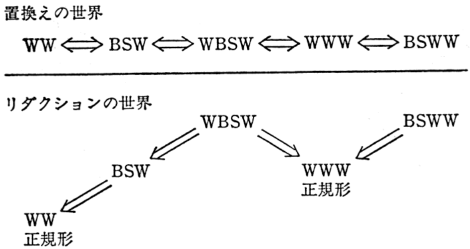


図 7.3 グラス列の置換えとリダクション

と方向づけてやると，可逆的な置換え規則Gから非可逆的なリダクション規則Rが得られる．

$$G \begin{cases} \mathrm{B} \leftrightarrow \mathrm{WB} \\ \mathrm{BS} \leftrightarrow \mathrm{W} \end{cases} \quad \Rightarrow \quad R \begin{cases} \mathrm{WB} \rightarrow \mathrm{B} \\ \mathrm{BS} \rightarrow \mathrm{W} \end{cases}$$

リダクション規則Rを用いると，グラス列は置換えによって，より簡単な列に変形されていく．たとえば，グラス列 WBSW は WBSW⇒BSW⇒WW と変形される．このような変形を**リダクション**と呼ぶ．また，WW のようにこれ以上変形ができないグラス列を**正規形**と呼ぶ．このとき，WW は WBSW の正規形であるという．WBSW↓ によって WBSW の一つの正規形（たとえば WW）を表現する．

グラス置換え規則に方向づけをしてリダクション規則を作ってやると，どちら方向へも変形できるグラスの置換えが，後戻りのできないリダクションをジグザグにつないだ形で表現される（図 7.3）．つまり，リダクション規則によってグラスの置換え方向に重みを付け，平面的な世界からデコボコした世界に問題を移し換えたわけである．このように移し換えたからといってパズルが必ずしもすぐに解けるわけではないが，たまたま移し換えた先が完備なリダクションシステムになっていると話は違ってくる．パズルは非常に簡単に解けてしまうのだ．それでは完備なシステムとはどのようなものかを次に説明しよう．

7.2.3 完備なシステム

適当なリダクション規則Rによってリダクション⇒が定められているなら，Rを**リダクションシステム**と呼ぶ．このとき，リダクションシステムRが**停止**

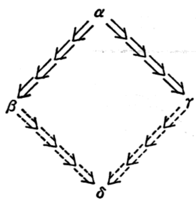


図 7.4　合流性

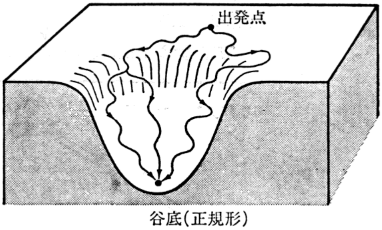


図 7.5　完備なリダクションシステム

性を満たすとは，無限のリダクション $\alpha_0 \Rightarrow \alpha_1 \Rightarrow \alpha_2$ …が存在しないことである．つまり，どのようなリダクションも必ず有限のステップで正規形に到達し，そこで止まってしまう．リダクションの世界をデコボコの世界だと考えれば，谷の深さはいつでも有限ということになる．

もう一つの重要な性質は**合流性**である．リダクションシステムRが合流性を満たすとは，α から2通りのリダクションによって β, γ が得られても，その後βとγに適当なリダクションをほどこすことで，ある δ に合流させることが必ずできることである (**図 7.4**)．デコボコの世界で考えると，出発点が同じなら途中のリダクションの道は異なっていても，結局は同じ谷を降りて行き，どこかで出会えることになる．

さて，リダクションシステム Rが停止性と合流性を満たすなら，Rは**完備**であるという．デコボコの世界で考えると，出発点が同じなら，すべてのリダクションは同じ谷を降りて行き，しかも谷の深さはいつでも有限なので，いつかは谷底で合流するということになる (**図 7.5**)．

リダクションシステムRが完備なら次の重要な二つの性質を持つことが知られている．

性質 1　αの正規形 $\alpha\downarrow$ は唯一に定まる．

これは，図7.5から直観的に明らかであろう．次に，グラス置換え規則Gから完備なリダクションシステムRが得られたとしよう．このとき，

性質 2　Gのもとで $\alpha \overset{*}{\Longleftrightarrow} \beta \Longleftrightarrow R$ のもとで $\alpha\downarrow \equiv \beta\downarrow$.

つまり，αとβ がグラス置換え規則 Gによって結べるということは，αとβ の正規形が一致するということと同値になる．(ここで，$\alpha\downarrow \equiv \beta\downarrow$ は $\alpha\downarrow$ と $\beta\downarrow$ が

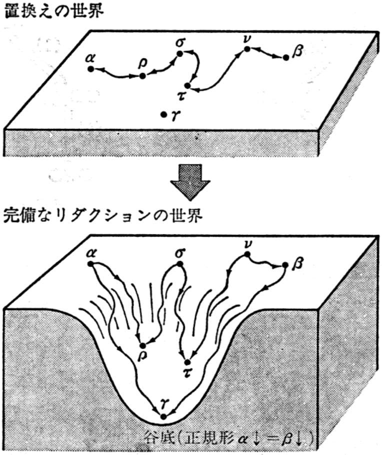


図 7.6 性質2の説明

まったく同じ形であることを示す.) 平面的な G の世界で α と β が結ばれるということは, デコボコな R の世界では α と β が登り降りする道で結ばれるということであるが, 谷底まで降りてしまえば最後には一点に合流してしまうわけである (**図 7.6**).

性質2を用いると, グラス置換えパズルは容易に解くことができる. $\alpha \overset{*}{\Longleftrightarrow} \beta$ が成立するか否かを問われたら, まずそれぞれの正規形 $\alpha\downarrow$ と $\beta\downarrow$ を求める. 完備なシステムでは谷の深さが有限であるから $\alpha\downarrow$, $\beta\downarrow$ は必ず求まる. ここで, $\alpha\downarrow \equiv \beta\downarrow$ なら答は **YES**, $\alpha\downarrow \not\equiv \beta\downarrow$ なら **NO** となる.

したがって, 置換え規則 G から完備なリダクションシステム R が得られたならば, 先ほどのパズルは解ける. しかし, 実際に得られた R は残念ながら完備ではない. 停止性を満たすことは明らかであるが, 図 7.3 で示したように WBSW が二つの正規形 WW と WWW を持つから合流性は満たされない. したがって, この R を直接用いてパズルを解くことはできない.

それでは, R がこのように合流性を満たさない原因はどこにあるのだろうか. 合流性が問題となるのは, 一つのグラス列に二つのリダクション規則を適用して別々のグラス列が得られる場合である. この状況を, 二つの規則の適用

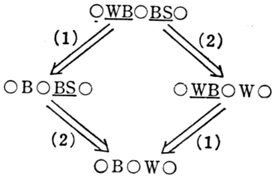


図 7.7　重ならない場合

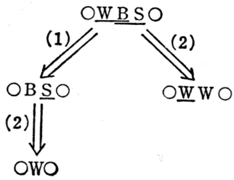


図 7.8　重なる場合

が**重ならない場合**と**重なる場合**に分けて少し詳しく検討してみよう．

まず，二つのリダクション規則 WB→B と BS→W の適用が重ならない場合を考えてみる．たとえば，○WB○BS○ なるグラス列を考える．ここで○は任意のグラスとする．このとき，このグラス列に適用可能な二つのリダクション規則の左辺 WB と BS は離れた場所に出現しているので，二つの規則は互いに独立に適用することができる．したがって，一方の規則の適用後でも，他方の規則のそれとは無関係に適用可能であり，リダクションは常に合流させることができる（図7.7）．

次に二つのリダクション規則の適用が重なる場合を考えてみよう．たとえば，グラス列○WBS○を考える．すると，二つの規則の左辺 WB と BS はグラスBを共有しており，この部分では重なりを持つ．したがって，一方の規則の適用は，他方の規則の適用を不可能にしてしまい，リダクションは合流できない（図7.8）．

以上のように，二つの規則に重なりが生ずると合流性が無条件には成立しなくなる．たとえば，二つの規則の左辺の重なり WBS にそれぞれの規則を適用すると BS と WW が得られる．しかし，図7.8で示したように一般にこれらは合流しない．ここで仮に，BS と WW がうまく合流できれば，R は合流性を満たすはずである．つまり，BS と WW は R が合流性を満たすか否かの鍵となっている．そこで，このような対 <BS, WW> を**危険対**と呼ぶ．危険対の要素をそれぞれリダクションして正規形を求めたとき，両者が一致しないなら危険対は発散するという．先ほどの例では $BS\downarrow \neq WW\downarrow$ であるから危険対 <BS, WW> は発散している．

このとき**停止性を満たすリダクションシステム R が完備であるための必要十**

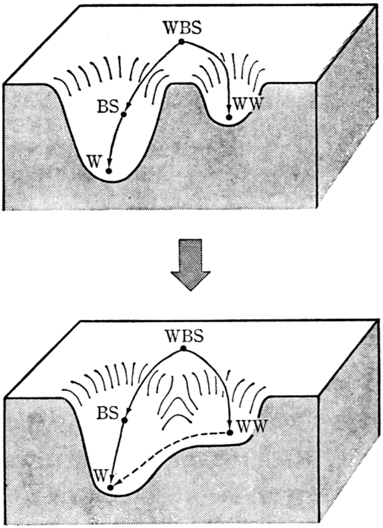


図 7.9 完備化の方法

分条件は R が発散する危険対を持たないことであることが知られている．したがって，R を完備にするためには発散する危険対をうまく消してやればよい．このためには以下で説明する完備化を行なう．

7.2.4 完備化

デコボコの世界で考えると，同じ点から出発して異なる二つの谷底へ到達するリダクションが，発散する危険対ということになる．そこで，谷底をうまく結ぶ道を付けて，二つの谷を一つにしてやる（**図 7.9**）．たとえば，危険対 <BS, WW> は二つの谷底 BS↓≡W と WW↓≡WW を持つ．そこでリダクション規則 WW→W を R に付け加えて谷底を一つにしてしまう．すると新しいリダクションシステム $\tilde{R}$ が得られる．

$$\tilde{R}\begin{cases}\mathrm{WB}\to\mathrm{B} & (1)\\ \mathrm{BS}\to\mathrm{W} & (2)\\ \mathrm{WW}\to\mathrm{W} & (3)\end{cases}$$

ここで注意してほしいのは，$\tilde{R}$ の世界で α と β がジグザグにつなげるなら，もとの G の世界でも $\alpha \overset{*}{\Longleftrightarrow} \beta$ となることである．つまり，$\tilde{R}$ を両方向の置換え規則

$$\tilde{G}\begin{cases} \mathrm{WB} \leftrightarrow \mathrm{B} \\ \mathrm{BS} \leftrightarrow \mathrm{W} \\ \mathrm{WW} \leftrightarrow \mathrm{W} \end{cases}$$

とみなすと，

G のもとで $\alpha \overset{*}{\Longleftrightarrow} \beta \Longleftrightarrow \tilde{G}$ のもとで $\alpha \overset{*}{\Longleftrightarrow} \beta$

が成立する．これは少し考えてみればあたりまえのことである．$\tilde{G}$ で新しく付け加わった置換え規則 WW↔W は，G によって WW⟺WBS⟺BS⟺W のようにシミュレーションできるからである．したがって，G の代わりに $\tilde{G}$ のもとでグラス置換えパズルを解いてもかまわない．

さて，新しく得られたリダクションシステム $\tilde{R}$ が完備であるか否かを調べてみよう．停止性については問題はないから，合流性についてのみ調べればよい．$\tilde{R}$ の危険対は以下の三つである．

（i）　<BS, WW>：WBS に規則(1)と(2)を適用

（ii）　<WB, WB>：WWB に規則(3)と(1)を適用

（iii）　<WW, WW>：WWW に規則(3)と(3)を適用

ここで（iii）は規則(3)を自分自身に位置をずらせて重ねたものである．発散する可能性のある危険対は（i）のみであるが，BS↓≡W≡WW↓ より，（i）も合流することがわかる．したがって，$\tilde{R}$ は発散する危険対を持たず，完備であることが示された．

以上の例では，R に新しい規則(3)を付け加えるだけで完備な $\tilde{R}$ が得られた．しかし，一般には新しい規則を付け加えることによって，新たな発散する危険対が生ずる可能性もある．したがって，そのような場合には発散する危険対が完全になくなるまで前述の手続きを繰り返す必要がある．また，リダクション規則の両辺がほかの規則によってリダクションできるなら，両辺をリダクションして正規形にしたほうがシステムは簡単になる．このような点まで考慮した完備化アルゴリズムを図7.10に示す．ここで＞は停止性を保証するための適当な順序であり，$\tilde{R}$ のすべての規則 $\alpha \to \beta$ が $\alpha > \beta$ を満たすなら $\tilde{R}$ は停止

完備化アルゴリズム

$\tilde{R}$ はリダクション規則の場合，G は置換え規則の場合，$>$は停止性を保証する順序である．
危険対 $<\alpha,\beta>$ はGの中では置換え規則 $\alpha\leftrightarrow\beta$ とみなす．

G が空でないなら以下を繰り返せ．もし G が空なら完備化は**成功**．

（1） G から $\alpha>\beta$ を満たす置換え規則 $\alpha\leftrightarrow\beta$（あるいは $\beta\leftrightarrow\alpha$）を一つ取り除く．もし，そのような規則がないなら完備化は**失敗**．

（2） $\tilde{R}$ のリダクション規則のうち左辺を $\alpha\to\beta$ でリダクションできるものはすべて G に移す．

（3） $\alpha\to\beta$ と $\tilde{R}$ によって生じたすべての危険対を G に付け加える．

（4） $\alpha\to\beta$ を $\tilde{R}$ に付け加える．

（5） $\tilde{R}$ を用いて $\tilde{R}$ のリダクション規則の右辺をすべて正規形にする．

（6） $\tilde{R}$ を用いて G の置換え規則の両辺をすべて正規形にする．両辺が一致するものはGから取り除く．

図 7.10　完備化アルゴリズム

するものとする．アルゴリズムは最初 $\tilde{R}=\phi$ で実行を開始する．そして，$G=\phi$ になれば完備化は成功で，$\tilde{R}$ は完備なリダクションシステムになる．なお，このアルゴリズムはループを永久に回り続けて停止しない場合もあるので正確には半アルゴリズムである．しかし，一般の呼び方に従ってここでは完備化アルゴリズムと呼ぶことにする．

完備なリダクションシステム $\tilde{R}$ の規則 $\alpha\to\beta$ を両方向の置換え規則 $\alpha\leftrightarrow\beta$ とみなすと，置換えシステム $\tilde{G}$ が得られる．すると完備化アルゴリズムは G から $\tilde{G}$ への変換アルゴリズムとみなすこともできる．つまり，与えられた G を変換して以下の 2 条件を満たす $\tilde{G}$ を求めるわけである（**図 7.11**）．

（1） Gのもとで $\alpha\overset{*}{\Longleftrightarrow}\beta \longleftrightarrow \tilde{G}$ のもとで $\alpha\overset{*}{\Longleftrightarrow}\beta$

（図 7.11 ではこれを $\underset{G}{\overset{*}{\Longleftrightarrow}} = \underset{\tilde{G}}{\overset{*}{\Longleftrightarrow}}$ で表わす．）

（2） $\tilde{G}$から得られるリダクションシステム $\tilde{R}$ は完備

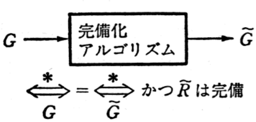


図 7.11　完備化アルゴリズムによる変換

7.2.5　パズルの答

それではグラス置換えパズルの答を示そう．与えられたグラス置換え規則Gから完備なリダクション規則$\tilde{R}$はすでに求めてある．このとき，Gと$\tilde{G}$は等価であることおよび完備な$\tilde{R}$の性質2を用いて，

$$G\text{のもとで}\ \alpha \overset{*}{\Longleftrightarrow} \beta \longleftrightarrow \tilde{G}\ \text{のもとで}\ \alpha \overset{*}{\Longleftrightarrow} \beta$$
$$\longleftrightarrow \tilde{R}\ \text{のもとで}\ \alpha\downarrow \equiv \beta\downarrow$$

が成立する．したがって，$\tilde{R}$を用いて正規形$\alpha\downarrow$，$\beta\downarrow$を求め，ついで$\alpha\downarrow \equiv \beta\downarrow$となるか否かを調べれば問題1は解ける．結果は以下のとおりである．

（1）　BSBS$\overset{*}{\Longleftrightarrow}$WWWB は成立しない．なぜなら，

BSBS$\downarrow \equiv$ W $\not\equiv$ B $\equiv$ WWWB$\downarrow$

（2）　BSBBS$\overset{*}{\Longleftrightarrow}$BWW は成立する．なぜなら，

BSBBS$\downarrow \equiv$ BW $\equiv$ BWW$\downarrow$

物理学では座標変換をほどこすことによって問題がいちじるしく簡単になることがある．たとえば，直交座標から極座標に変換することで波動方程式がうまく解ける場合などがそうである．完備化もこの点では座標交換と似ている．与えられたGを完備な$\tilde{G}$(つまり$\tilde{R}$) に変換することによって，Gのもとでは容易に解けなかった問題がきわめて簡単に解けてしまう．この座標変換テクニックは，もちろんグラス置換えパズルだけではなく，さまざまな対象に適用できる．以下では，普遍代数，多項式イデアル理論，一階述語論理における完備化アルゴリズムを紹介しよう．

7.3　普遍代数における完備化アルゴリズム

ここでは普遍代数を対象とした完備化アルゴリズムについて説明しよう．このアルゴリズムは普遍代数の語の問題を解くため，Knuth と Bendix によって 1967 年に発表された． Knuth-Bendix 完備化アルゴリズムについては，すでに発行済みの『新しいプログラミング・パラダイム』第8章「項書換えシステムと完備化手続き」[5] で大須賀昭彦氏が大変わかりやすく紹介しているので，詳細は大須賀氏の解説にまかせ，ここでは基本的な考え方のみを述べることにする．

7.3.1 群論の語の問題

Knuth と Bendix が完備化アルゴリズムを発見するきっかけとなった群論の語の問題を考えてみよう．よく知られているように群論は次の公理Eによって定められる．

$$E\begin{cases} e\cdot x=x & \text{(左単位元 } e \text{ の存在)} \\ x^{-1}\cdot x=e & \text{(左逆元の存在)} \\ (x\cdot y)\cdot z=x\cdot(y\cdot z) & \text{(結合律)} \end{cases}$$

（つまり，群論では変数 x, y, z に何が代入されても，Eの等式は成立する．）このとき，公理 E のもとで等式 $t=s$ が成立するか否かを決定する問題を群論の語の問題という．たとえば，E のもとで $x^{-1}\cdot(x\cdot x)=x$ が成立することは以下のように証明できる．

$$x^{-1}\cdot(x\cdot x)=(x^{-1}\cdot x)\cdot x=e\cdot x=x$$

それでは次の問題はどうだろうか．

問題 2　Eのもとで $e\cdot x=x\cdot e$ は成立するか．

代数の教科書に演習としてよく取り上げられる問題であるが実際に解かれた読者は少ないと思う．直観的には成立するのが当然にみえる等式であるが，公理Eのみを用いて左辺と右辺が等しいことを示すのは意外に大変なのである．（以前に何人かでこの問題に挑戦したときには，1時間以内にこの等式を証明できたのは全体の半数であった．）

したがって，計算機を用いてこの問題を解く場合にも，公理Eをやみくもに使って左辺と右辺をなんとか等号で結び付けようとするのは，あまりうまい方法ではない．試行錯誤のループからなかなか抜け出せないばかりか，場合によっては方向違いなことを証明しようとして永久に走り続けることにもなりかねない．また，与えられた等式がEで成立しない場合に，どうやって成立しないことを証明するのかという困難な問題も残っている．

7.3.2 群の公理の完備化

ここで前節で述べたグラス置換えパズルを思い出そう．群の語の問題とグラス置換えパズルがよく似た構造を持つことに気づくであろう．公理Eはグラス

置換え規則 Gに対応し，E のもとで等式 $t=s$ が成立するか否かは，G のもとで $\alpha \overset{*}{\Longleftrightarrow} \beta$ が成立するか否かに対応する．そこで，このアナロジーを押し進めるために，公理 E からリダクションシステム R を作ってみよう．両方向の書換えが可能な公理 $a=b$ を，$a \to b$ と方向づけてやれば，リダクションシステム Rが前節と同様に得られる．

$$E \begin{cases} e \cdot x = x \\ x^{-1} \cdot x = e \\ (x \cdot y) \cdot z = x \cdot (y \cdot z) \end{cases} \Rightarrow \quad R \begin{cases} e \cdot x \to x & (1) \\ x^{-1} \cdot x \to e & (2) \\ (x \cdot y) \cdot z \to x \cdot (y \cdot z) & (3) \end{cases}$$

（ただし，変数 x，y，z には何が代入されてもよいから，たとえば，任意の項 t に対して $e \cdot t \Rightarrow t$ なるリダクションが Rの規則(1)によって得られる．）

このように，等式を方向づけて得られたリダクションシステム Rは**項書換えシステム**と呼ばれる．

次のステップはRの完備化である．完備化を行なうためには前節で述べたように危険対を求める必要がある．そこでRの書換え規則の重なりを調べてみよう．たとえば，項 $(x^{-1} \cdot x) \cdot y$ を考える．すると，規則(2)と(3)から発散する危険対 $\langle e \cdot y, x^{-1} \cdot (x \cdot y) \rangle$ が得られる（**図7.12**）．（今度は，リダクション規則の変数には何が代入されてもよいということに注意しよう．）したがって，発散する危険対をなくすためには $x^{-1} \cdot (x \cdot y) \to y$ を新しい規則として R に付け加える必要がある．このようにして，グラス置換えパズルとまったく同様な手続きで項書換えシステムRの完備化を行なうことができる．以下で述べる Knuth–Bendix 完備化アルゴリズムを用いて完備な項書換えシステム $\tilde{R}$ を求めると最終的には次の 10 個の書換え規則が得られる．

$$\tilde{R} \begin{cases} e \cdot x \to x & (1) \\ x^{-1} \cdot x \to e & (2) \\ (x \cdot y) \cdot z \to x \cdot (y \cdot z) & (3) \\ x^{-1} \cdot (x \cdot y) \to y & (4) \\ x \cdot e \to x & (5) \\ e^{-1} \to e & (6) \\ (x^{-1})^{-1} \to x & (7) \\ x \cdot x^{-1} \to e & (8) \\ x \cdot (x^{-1} \cdot y) \to y & (9) \\ (x \cdot y)^{-1} \to y^{-1} \cdot x^{-1} & (10) \end{cases}$$

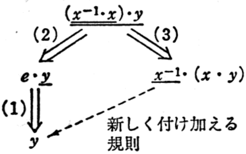


図 7.12 R の危険対

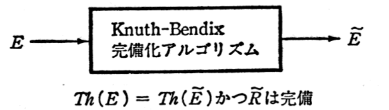


図 7.13 Knuth-Bendix 完備化アルゴリズムによる変換

これで群論の語の問題は完全に解けたことになる．なぜなら前節で述べたように以下が成立するからである．

E のもとで $t=s$ が成立 $\Longleftrightarrow$ $\tilde{R}$ のもとで $t\downarrow\equiv s\downarrow$（ただし $t\downarrow\equiv s\downarrow$ は $t\downarrow$ と $s\downarrow$ が構文的に一致していることを示す.）

このとき問題 2 の答は $e\cdot x\downarrow\equiv x\equiv x\cdot e\downarrow$ より **YES** である．

7.3.3 Knuth-Bendix 完備化アルゴリズム

与えられた等式の集合 E から完備な項書換えシステム $\tilde{R}$ を求める Knuth-Bendix 完備化アルゴリズムは，グラス置換えパズルの完備化アルゴリズム（図 7.10)をほんの少し修正するだけで得られる．修正する部分は次のとおりである．

「置換え規則の集合 G」⇒「等式の集合 E」

「置換え規則 $\alpha\leftrightarrow\beta$」 ⇒「等式 $\alpha=\beta$」

このように修正するだけで，図 7.10 の完備化アルゴリズムが等式の集合 E に対してもうまく働くことは明らかであろう．

完備な項書換えシステム $\tilde{R}$ の書換え規則 $t\rightarrow s$ を等式 $t=s$ とみなすと，等式の集合 $\tilde{E}$ が得られる．前節で述べたように，

E のもとで $t=s$ $\Longleftrightarrow$ $\tilde{E}$ のもとで $t=s$

であるから，公理 E から得られる定理の集合 $Th(E)$ と $\tilde{E}$ から得られる $Th(\tilde{E})$ は一致する．したがって，Knuth-Bendix 完備化アルゴリズムは図 7.13 のような変換アルゴリズムとみなすこともできる．

7.4 多項式イデアル理論における完備化アルゴリズム

次に多項式イデアルを対象とした完備化アルゴリズムについて説明しよう．

このアルゴリズムは多項式イデアルの Gröbner (グレブナ) 基底を計算するために Buchberger によって1965年に考案された．しかし，Buchberger アルゴリズムが完備化アルゴリズムであることが広く認識されるようになったのは1980年代後半になってからである．以下ではまず Buchberger アルゴリズムの特殊な場合であるユークリッドの互除法から出発しよう．

7.4.1　ユークリッドの互除法

与えられた自然数 m, n の最大公約数 d を求めるためのユークリッドの互除法はよく知られたアルゴリズムである．ここではユークリッドの互除法がなぜ完備化アルゴリズムに関係あるのかを述べよう．

まず，**イデアル**について簡単に説明する．$\boldsymbol{Z}$ を整数集合とする．このとき $c_1, c_2, \cdots, c_n \in \boldsymbol{Z}$ から生成されるイデアル I とは

$$I=(c_1, c_2, \cdots, c_n)=\{c_1a_1+c_2a_2+\cdots+c_na_n \mid a_i \in \boldsymbol{Z}\}$$

で定められる $\boldsymbol{Z}$ の部分集合である．このとき $c_1, c_2, \cdots, c_n$ は**イデアル基底**と呼ばれる．たとえば，

$$(3)=\{3a \mid a \in \boldsymbol{Z}\}$$

$$(2,6)=\{2a+6b \mid a, b \in \boldsymbol{Z}\}$$

となる．ここで(3)のように1個の基底3から生成されるイデアルは単項イデアルと呼ばれる．なお，イデアル基底の要素をすべて正整数に制限しても一般性は失われないので，以下では $c_i>0$ と仮定して話を進める．

二つの整数 m, n がイデアル I を法として合同であるとは $m-n \in I$ なることであり，このとき $m=n(\mathrm{mod}\ I)$ と書く．これは通常の合同式の一般化になっている．

ここで次の問題を考えてみよう．

問題3　$I=(21, 6)$ とする．このとき $8=5(\mathrm{mod}\ I)$ は成立するか．

答は **YES** である．というのは以下の便利な性質が利用できるからである．

イデアル基底 $c_1, \cdots, c_n$ の最大公約数 $\mathrm{GCD}(c_1, \cdots, c_n)=d$ ならば，$(c_1, \cdots, c_n)=(d)$ となる．つまり，$c_1, \cdots, c_n$ で生成されるイデアルは d で生成される単項イデアルと等しくなる．よって，$m=n(\mathrm{mod}(c_1, \cdots, c_n))$ と $m=n(\mathrm{mod}\ (d))$ は等価とい

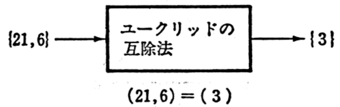


図 7.14　ユークリッドの互除法による変換

うことになる．

GCD(21, 6)＝3 であるから，上記の性質を用いると 8≡5(mod(21, 6)) の代わりに 8≡5(mod(3)) を考えてもかまわない．これが成立することは8−5＝3より明らかであろう．

このように，最大公約数を求めることでイデアル基底 {21, 6} からより簡単な {3} に変換できる．このとき，それぞれの基底によって生成されるイデアルは等しい．最大公約数はユークリッドの互除法によって求めることができるから，以上の関係は図 7.14 のように表わせる．この図には完備化の香りがほのかではあるが漂っているであろう．

それでは，ユークリッドの互除法の中身を見てみよう．自然数 $m, n (m \geq n)$ に対して最大公約数 GCD(m, n) は以下のように定められる．

$$\mathrm{GCD}(m, n) = \begin{cases} m : n=0 \text{ の場合} \\ \mathrm{GCD}(n, m \bmod n) : n \neq 0 \text{ の場合} \end{cases}$$

ここで $m \bmod n$ は m を n で割った余りである．このとき $<m, n>$ よりは $<n, m \bmod n>$ のほうが辞書式順序のもとで小さくなっていることに注意する．したがって，再帰的に GCD を計算すれば最後に最大公約数が求まる．これがユークリッドの互除法である．実際に 21 と 6 の最大公約数を求めてみよう．

$$\mathrm{GCD}(21, 6) = \mathrm{GCD}(6, 3) = \mathrm{GCD}(3, 0) = 3$$

ユークリッドの互除法を完備化アルゴリズムとみなすためには，リダクションシステム R を明らかにしなければならない．そこで，イデアル基底 21 と 6 から以下のリダクションシステム R を作ってみよう．

$$R \begin{cases} 21 \to 0 & (1) \\ 6 \to 0 & (2) \end{cases}$$

ここで，自然数 $m = pn + r (n > r \geq 0)$ はリダクション規則 $n \to 0$ を用いて $m \Rightarrow r$ とリダクションできることにする．たとえば，14＝2・6＋2 であるから規則 (2)

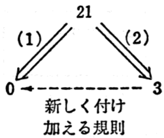


図 7.15　R の危険対

を適用するとリダクション $14 \Rightarrow 2$ が得られる．R は明らかに停止性を満たしている．しかし合流性は満たさない．なぜなら，$21=3\cdot6+3$ であるから規則(1)と(2)は重なり，発散する危険対を持つ（図 7.15）．ここで R に新しい規則 $3 \rightarrow 0$ を付け加える．

$$R' \begin{cases} 21 \rightarrow 0 & (1) \\ 6 \rightarrow 0 & (2) \\ 3 \rightarrow 0 & (3) \end{cases}$$

このとき規則(2)と(3)を用いて $21 \overset{*}{\Rightarrow} 0$ が得られる．ここで $\overset{*}{\Rightarrow}$ は0回以上のリダクションを表わす．よって冗長な規則(1)は取り除いてもかまわない．すると R'' が得られる．

$$R'' \begin{cases} 6 \rightarrow 0 & (2) \\ 3 \rightarrow 0 & (3) \end{cases}$$

これはまさに GCD(21, 6) から GCD(6, 3) への変形に対応しているだろう．この手続きを繰り返して最後に完備なリダクションシステム

$$\tilde{R}\ \{3 \rightarrow 0 \qquad (3)$$

が得られる．以上でユークリッドの互除法が完備化アルゴリズムであることは理解できたと思う．当然

$$8=5(\mathrm{mod}(21, 6)) \Longleftrightarrow \tilde{R} \text{ のもとで } 8\downarrow\equiv5\downarrow$$

となるから，$8\downarrow\equiv2\equiv5\downarrow$ を用いて $8=5(\mathrm{mod}(21, 6))$ が示される．

グラス置換えパズルの完備化アルゴリズム（図7.10）を以下のように修正すれば，ユークリッドの互除法が得られる．

「置換え規則の集合 G」$\Rightarrow$「イデアル基底 B」

「置換え規則 $\alpha\leftrightarrow\beta$」　$\Rightarrow$「等式 $\alpha=\beta$」

ここで，イデアル基底の要素 α は形式的に等式 $\alpha=0$ で表現されているものと約束する．したがって，図 7.10 の完備化アルゴリズムで β は常に 0 となる．

7.4.2 多項式イデアル

有理係数の一変数多項式，$5x^4-\dfrac{1}{2}$，$2x^3-\dfrac{1}{3}x+6$ などの集合を $\boldsymbol{Q}[x]$ と書く．このとき，$\boldsymbol{Q}[x]$ 上での四則演算は整数 $\boldsymbol{Z}$ 上と同様に定められる．したがって，$\boldsymbol{Z}$ 上と同じようにしてイデアルを考えることができる．イデアル基底 $f_1, \cdots, f_n \in \boldsymbol{Q}[x]$ から生成されるイデアル I は

$$I=(f_1, \cdots, f_n)=\{f_1u_1+\cdots+f_nu_n \mid u_i \in \boldsymbol{Q}[x]\}$$

で定められる．また，二つの多項式 $g, h \in \boldsymbol{Z}[x]$ が $g=h \pmod I$ とは $g-h \in I$ となることである．

$\boldsymbol{Q}[x]$ 上では $\boldsymbol{Z}$ 上と同様に最大公約数も定義でき，ユークリッドの互除法によってそれを求めることもできる．したがって，$\boldsymbol{Z}$ の場合と同様にユークリッドの互除法による完備化を利用して合同式を解くことが可能である．以下の問題を通してこれを示そう．

問題 4 $I=(3x^3-4x+1, x^2-4x+3)$，$g(x)=x^3+3x^2-8$，$h(x)=x^3+2x^2-7$ とする．このとき $g(x)=h(x) \pmod I$ は成立するか．

答は **YES** である．前述した問題 3 の解法と同様に，ユークリッドの互除法で $\mathrm{GCD}(3x^2-4x+1, x^2-4x+3)=x-1$ を求める．ここで先ほどは $x-1 \to 0$ をリダクション規則とおいた．しかし，今回はもう少し使いやすくするために，最も複雑な項を左辺に残し，残りは右辺に移項して次の規則に変形する．

$$\tilde{R}\ \{x \to 1$$

すると $\tilde{R}$ のもとで $g(x) \overset{*}{\Rightarrow} 1^3+3\cdot1^2-8=-4$，$h(x) \overset{*}{\Rightarrow} 1^3+2\cdot1^2-7=-4$ より $g(x)\downarrow \equiv -4 \equiv h(x)\downarrow$ が得られる．よって，$g(x)=h(x) \pmod I$ は示された．

7.4.3 Buchberger アルゴリズム

今度は二つの変数 x，y を持つ多変数多項式の集合 $\boldsymbol{Q}[x, y]$ で同様な問題を考えてみよう．(以下の問題は文献 7) から借用した．)

問題 5 $f_1=x^3y+xy+1$，$f_2=x^2y^2-y^2+1$，$I=(f_1, f_2)$ とする．$g=2xy^2+2y^2+y$，$h=x^2-xy+x+2$ とすると，$g=h \pmod I$ は成立するか．

前述した方法に従えば $\mathrm{GCD}(f_1, f_2)=f$ を求め，f のもとで $g\downarrow\equiv h\downarrow$ が成立するか否かを調べればよい．ところが残念ながら，この方法を今回は使うことができない．一変数多項式に対しては，整数に対する議論をほとんどそのまま繰り返すことによって，最大公約数やユークリッドの互除法を考えることができた．ところが，多変数多項式に対しては，このようなアナロジーは成立しないことが知られている．(正確に述べると，$\boldsymbol{Q}[x]$ はユークリッド環なのに $\boldsymbol{Q}[x, y]$ はそうでないということである．) そこで Buchberger は最大公約数の代わりに **Gröbner** (グレブナ) **基底**を考え，与えられたイデアル基底から Gröbner 基底を計算するためのアルゴリズムを提案した．これが Buchberger アルゴリズムである．このアルゴリズムに従って問題を解いてみよう．

まずイデアル基底 f_1，f_2 からリダクション規則を作らなければならない．基底の要素を形式的に $f_1=0$，$f_2=0$ の等式で表わすと，リダクション規則は $f_1\rightarrow 0$，$f_2\rightarrow 0$，つまり $x^3y+xy+1\rightarrow 0$，$x^2y^2-y^2+1\rightarrow 0$ となる．しかし，この形では使いにくい．そこで，最も複雑な項を左辺に残し，残りは右辺に移項してやる．

$$R\left\{\begin{array}{ll} x^3y \rightarrow -xy-1 & (1)\\ x^2y^2 \rightarrow y^2-1 & (2)\end{array}\right.$$

(ただし，ここでは x, y, z を定数とみなし，代入は許さない．たとえば，規則 (1) を用いて $z^3x \Rightarrow -zx-1$ なるリダクションを行なうことはできない．)

以下でも同様に，等式からリダクション規則を作る場合には，最も複雑な項を左辺，残りは右辺に移項して規則を作ることに約束しておく．

次に R の完備化を行なう．このとき，規則 (1) と (2) は項 x^3y^2 で重なり，発散する危険対 $\langle -xy^2-y, xy^2-x\rangle$ を持つ (図 7.16)．ここで危険対を等式 $-xy^2-y=xy^2-x$ とみなし．この等式から書換え規則 $2xy^2\rightarrow x-y$ を作る．この規則を R に付け加えると R' が得られる．

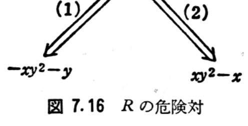


図 7.16　R の危険対

$$R' \begin{cases} x^3y \to -xy-1 & (1) \\ x^2y^2 \to y^2-1 & (2) \\ 2xy^2 \to x-y & (3) \end{cases}$$

以下同様な手続きを繰り返すと最後に完備なリダクションシステム $\tilde{R}$ が得られる．

$$\tilde{R} \begin{cases} x^3y \to -xy-1 & (1) \\ x^2y^2 \to y^2-1 & (2) \\ 2xy^2 \to x-y & (3) \\ x^2 \to xy+2y^2-2 & (4) \\ 4y^2 \to -xy+7x^2-2 & (5) \end{cases}$$

ここで $\tilde{R}$ のリダクション規則を等式と考え，すべての項を左辺に移項すると，形式的に等式 $f=0$ で表わされたイデアル基底の要素 f が得られる．

$$\tilde{B} \begin{cases} x^3y+xy+1 \\ x^2y^2-y^2+1 \\ 2xy^2-x+y \\ x^2-xy-2y^2+2 \\ 4y^2+xy-7x^2+2 \end{cases}$$

(ここでは等式 $f=0$ の右辺は取り除いてある.) この $\tilde{B}$ を Buchberger は Gröbner 基底と呼んだ．以上の手続きがまさに完備化アルゴリズムであることは容易に理解できるだろう．このとき，何度も述べてきたように，

$$g=h \pmod{I} \Longleftrightarrow \tilde{R} \text{ のもとで } g\downarrow \equiv h\downarrow$$

が成立する．問題 5 の場合には $g\downarrow \equiv 2y^2+x \equiv h\downarrow$ となるから答は **YES** である．

グラス置換えパズルの完備化アルゴリズム（図 7.10）から Buchberger アルゴリズムを得るためには以下の修正をほどこせばよい．

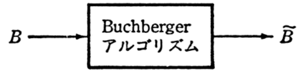
まず，イデアル基底の要素 f は形式的に等式 $f=0$ で表現されているものとする．また，等式からリダクション規則 $\alpha \to \beta$ を作る場合には，等式の中に出現している項のうち全順序＞のもとで最大の項を α とし，残りを移項して β とする．さらに，

「置換え規則の集合 G」⇒「イデアル基底 B」

「置換え規則 $\alpha \leftrightarrow \beta$」 ⇒「等式 $\alpha=\beta$」

とし，またアルゴリズムの (1) を以下のように修正する．

「(1) G から等式を一つ取り除き，その等式と全順序＞を用いてリダクショ



$I(B)=I(\widetilde{B})$ かつ $\widetilde{R}$ は完備

図 7.17　Buchberger アルゴリズムによる変換

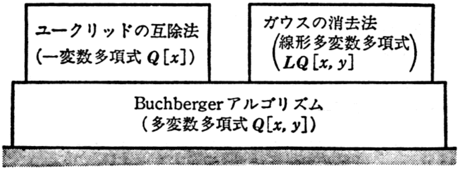


図 7.18　三つのアルゴリズムの関係

ン規則 $\alpha \to \beta$ を作る.」

Buchberger アルゴリズムはイデアル基底 B を Gröbner 基底 $\tilde{B}$ に変換するが，このときそれぞれの基底から得られるイデアル $I(B)$ と $I(\tilde{B})$ は一致する．図 7.17 にこの関係を示す.

多変数多項式 $Q[x, y]$ に対する Buchberger アルゴリズムは，当然その特殊な場合として一変数多項式 $Q[x]$ に対しても適用できる．これがユークリッドの互除法である．ほかの特殊な場合として線形多変数多項式 $LQ[x, y]$（つまり変数の次数は 1 以下）に適用すると，連立一次方程式の解法として知られているガウスの消去法が得られる（図 7.18）．つまり，連立一次方程式も実は完備化によって解かれていたわけである.

7.5　一階述語論理における完備化アルゴリズム

いよいよ最後に一階述語論理を対象とした完備化アルゴリズムについて述べよう．このアルゴリズムは定理自動証明のために 1965 年に Robinson によって考案されたレゾリューション法である．（レゾリューション法については，論理型言語 Prolog の計算メカニズムとして広く知られているので，ここでは詳しく説明はしない.）多くの研究者が Knuth-Bendix 完備化アルゴリズムとレゾリューション法の間にいくつかの共通点を発見していたにもかかわらず，レ

ゾリューション法が完備化アルゴリズムの一つであることは長い間見抜かれなかった．とくに，関数型言語と論理型言語の融合や，等式を含む一階述語論理の定理自動証明では，二つのアルゴリズムがさまざまなレベルで混在して利用されていたにもかかわらずである．両者の関係が明らかになったのは意外に遅く1985年になってからである．

このように発見が遅れた原因は，レゾリューション法が完備化アルゴリズムをふつうの使い方で利用していないからである．すでに何度も述べてきたように，完備化アルゴリズムは与えられたシステムを完備なシステムに変換する手続きである．しかし，レゾリューション法で得られるのは空節□のみであり，どこを見回しても完備なシステムらしいものは得られていない．それにもかかわらず，レゾリューション法が完備化になっていることを以下では説明しよう．（以下では簡単のためSLDレゾリューションを例にとって説明するが，一般のレゾリューション法も同様に完備化になっていることが知られている[4]．）

7.5.1 レゾリューション法

まず公理Aのもとで定理$\exists x \exists y S(x,y)$をレゾリューション法によって証明してみよう．

$$A \begin{cases} P(x,a) \leftarrow & (1) \\ Q(b,y) \leftarrow & (2) \\ S(x,y) \leftarrow P(x,y),\, Q(x,y) & (3) \end{cases}$$

レゾリューション法では証明すべき定理の否定と公理を用いて空節□（矛盾）を導けばよい．定理の否定は，

$$\neg \exists x \exists y S(x,y) \equiv \forall x \forall y \neg S(x,y) \equiv \neg S(x,y) \equiv \leftarrow S(x,y)$$

であるから，ゴール節$\leftarrow S(x,y)$から出発すればよい．**図7.19**にレゾリューションによる証明を示す．

さて，レゾリューションを完備化とみなすためには，まずリダクションシステムを定めなければならない．公理Aから自然に得られるリダクションシステムは，公理を左辺から右辺への書換え規則とみなした以下のシステムであろう．

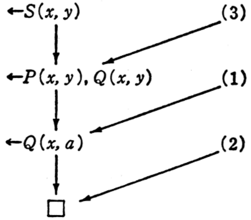


図 7.19　レゾリューションによる定理証明

$$R \begin{cases} P(x,a) \rightarrow \\ Q(b,y) \rightarrow \\ S(x,y) \rightarrow P(x,y),\, Q(x,y) \end{cases}$$

しかし，図 7.19 をいくらにらんでも，これがRの完備化とはとても思えない．そもそも，危険対はどれなのかさっぱりわからないのが致命的である．そんなわけで，ほとんどの研究者はレゾリューション法と完備化アルゴリズムは似て非なるものと考えたのであった．

7.5.2　Hsiang のアイデア

1985 年に Hsiang（シアン）は排他的論理和 $\oplus$（つまり $1\oplus 0=0\oplus 1=1$，$0\oplus 0=1\oplus 1=0$）を用いて公理 A を表現すれば，レゾリューション法と完備化アルゴリズムの自然な対応が得られることを発見した[3]．

Hsiang の方法では，まず真を 1，偽を 0 で表わす．そして，論理記号$\vee$, $\wedge$, $\neg$, $\leftarrow$を排他的論理和$\oplus$と論理積・のみで表現する．たとえば，$P\vee Q$ は $P\cdot Q\oplus P\oplus Q$（つまり $(P\cdot Q)\oplus P\oplus Q$ の意味）となるし，$\neg P$ は $P\oplus 1$ と表わせる．さらに，Pが真なら等式 $P=1$ で，偽なら $P=0$ で表わす．すると，$P\leftarrow Q$ は $P\cdot Q\oplus Q\oplus 1$ であるから，$P\leftarrow Q$ が真であることは，$P\cdot Q\oplus Q\oplus 1=1$ で表現できる．さらにこの等式は $P\cdot Q=Q$ と変形できる．したがって，$S\leftarrow P\cdot Q$ が真ならば，対応する等式は$S\cdot P\cdot Q=P\cdot Q$となる．

公理 Aをシアンの方法で表現すると次のようになる．

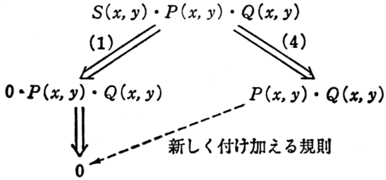
$$A \begin{cases} P(x,a)=1 \\ Q(b,y)=1 \\ S(x,y)\cdot P(x,y)\cdot Q(x,y)=P(x,y)\cdot Q(x,y) \end{cases}$$

これに証明すべき定理の否定 $\neg S(x, y)$ を真と仮定して付け加える. $\neg S(x, y)=1$ は $S(x, y)=0$ に変形できるから $S(x, y)=0$ を A に付け加えればよい. これらの等式を左辺から右辺へのリダクション規則とみなすと, $A \cup \{S(x, y)=0\}$ から以下のリダクションシステム R が得られる.

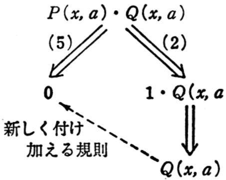
$$R \begin{cases} S(x, y) \to 0 & (1) \\ P(x, a) \to 1 & (2) \\ Q(b, y) \to 1 & (3) \\ S(x, y) \cdot P(x, y) \cdot Q(x, y) \to P(x, y) \cdot Q(x, y) & (4) \end{cases}$$

(このとき, 変数 x, y には何を代入してもよい. たとえば, 任意の項 t, s に対して $S(t, s) \Rightarrow 0$ なるリダクションが規則(1)から得られる.)

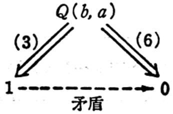
ここで R の規則の重なりを調べよう. まず規則(1)と(4)の重なりから危険対 $<0 \cdot P(x, y) \cdot Q(x, y),\ P(x, y) \cdot Q(x, y)>$ が得られる (図 7.20(a)). (ここで



(a)



(b)



(c)

図 7.20 R の完備化

$0 \cdot x \to 0$ などの規則は暗黙に仮定している.）この危険対から得られる規則(5)をRに付け加える.

$$P(x, y) \cdot Q(x, y) \to 0 \tag{5}$$

次に規則(5)と(2)の重なりから危険対を計算し（図7.20(b)），その結果得られた規則(6)をさらに付け加える.

$$Q(x, a) \to 0 \tag{6}$$

今度は規則(6)と(3)の重なりを調べてみよう(図7.20(c))．すると規則$1 \to 0$が得られる.

ここで，1が真，0が偽であることを思い出すと，この規則によって真と偽は一致してしまい矛盾が生ずる．これは，レゾリューション法によって空節□が得られたことに対応する．完備化手続きのほうは，1と0を一致させる規則が発見された時点で，矛盾が生じたということで失敗に終わる．つまり，レゾリューションによる定理証明を完備化の立場から見ると，途中で矛盾が生じたために完備化が失敗したことになる.

完備化アルゴリズムを座標変換と考えると，完備化の途中で矛盾が生ずるのは，三次元空間を表わす座標が変換の途中で二次元平面の座標に縮退してしまったようなものである．保存すべき構造が変換によって壊されてしまったわけである．その原因となったのは，証明すべき定理を否定して公理に付け加えたことである．つまり，本来は独立できない座標軸を，あたかも独立しているかのように取り扱い，ワザと座標変換を失敗に終わらせるのがレゾリューション法ということになる.

それでは，証明すべき定理の否定を付け加えずに，公理Aだけを完備化したらどうなるだろうか．当然のことながらこの場合には，完備化アルゴリズムは公理 A を公理 $\tilde{A}$ に変換する手続きになる．このとき，それぞれの公理から得られる定理の集合 $Th(A)$ と $Th(\tilde{A})$ は一致し，しかも $\tilde{A}$ から得られるリダク

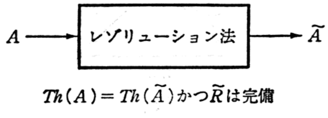


図 7.21　レゾリューション法による変換

ションシステム $\tilde{R}$ は完備となる（図 7.21）．このとき，論理式 P が定理か否かは $\tilde{R}$ を用いて決定できる．

$$P\text{が定理} \Longleftrightarrow A\text{のもとで } P=1$$
$$\Longleftrightarrow \tilde{R}\text{ のもとで } P{\downarrow}\equiv 1 \ (\equiv 1{\downarrow})$$

しかし，P が定理か否かという問題は一階述語論理では決定不能なはずである．どこにトリックがあったのだろうか．実は完備化が成功して A から $\tilde{A}$ が得られたと仮定したのが手品の種である．一階述語論理に対しては，完備化手続きが成功することはめったにない．ほとんどの場合，完備化手続きは無限ループから抜け出せずに永久に走り続けてしまう．（完備化アルゴリズムが半アルゴリズムであったことを思い出そう．）そこで，レゾリューション法では A の完備化をあきらめ，その代わりに $A\cup\{\neg P\}$ の完備化をワザと失敗させることによって定理証明を行なう．

最後に付け加えると，一階述語論理は二つの演算⊕と・についての線形多変数多項式環とみなすこともできる[4]．このとき，公理 A から $\tilde{A}$ への完備化アルゴリズムは，一階イデアル基底から Gröbner 基底を計算する Buchberger アルゴリズムとなっている．完備化の概念を通して論理構造と代数構造がこのように重なったわけである．

7.6 おわりに

普遍代数，多項式イデアル理論，一階述語論理の各分野で重要な役割を果たしている三つのアルゴリズム，Knuth-Bendix 完備化アルゴリズム，Buchberger アルゴリズム，レゾリューション法が，完備化アルゴリズムという一つの視点によって統一的に説明できることを示した（図 7.22）．ここで示した鳥瞰図から三つの世界がどのように結び付いていたのかを読み取っていただけたと思う．

1980年代後半に完備化という統一的概念が発見されて以来，この概念を核として各分野の理論研究や応用研究を融合する試みが活発に進められている．たとえば，条件付き項書換えシステムやナローイング手法を基礎にした関数・論理・融合型言語の研究，項書換えシステムを利用した一階述語論理の定理自動

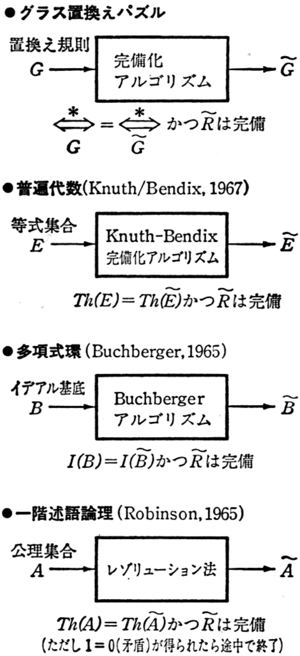


図 7.22　さまざまな完備化

証明の研究などは，普遍代数と一階述語論理の融合を目ざす最近のホットな話題である.

1965年前後にあいついで発見された三つのアルゴリズムが，実は一般化されたユークリッドの互除法，つまり完備化アルゴリズムの三つの異なる適用例であったことは偶然とは思えない．近い将来，第4，第5の完備化アルゴリズムの適用例が，予想もされなかった分野で発見されることは確実であろう．完備化の概念を最大公約数として現在形成されつつある新しいパラダイムが今後どのように発展していくか楽しみである.

最後に参考文献を挙げておこう．完備化アルゴリズムの歴史と全体像については文献1) をすすめる．Knuth-Bendix 完備化アルゴリズムについては文献5)

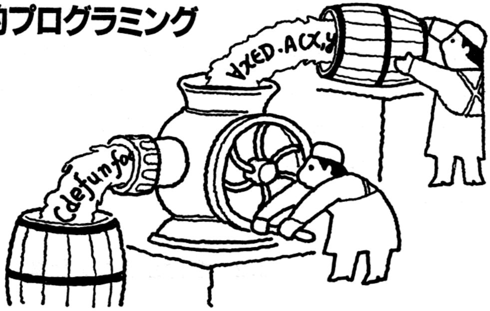
がわかりやすい．上級者向けには 6) がよくまとまっている．Buchberger アルゴリズムの入門としては，さまざまな応用例がていねいに説明してある 2) が最適である．簡単にエッセンスのみを知りたい方には 7) をすすめる．完備化による一階述語論理の定理自動証明については 3) を，またレゾリューション法と Buchberger アルゴリズムの関係については 4) をあげておく．

参考文献

1) B. Buchberger: History and basic features of the critical-pair/completion procedure, *J. Symbolic Computation 3*, pp. 54–71, 1987.
2) B. Buchberger: Gröbner bases: an algorithmic method in polynomial ideal theory, in: N. K. Bose, ed., *Recent Trends in Multidimensional Systems*, pp. 184–232, D. Reidel, 1985.
3) J. Hsiang: Refutational theorem proving using termrewriting systems, *Artificial Intelligence 25*, pp. 225–300, 1985.
4) D. Kapur and P. Narendran: An equational approach to theorem proving in first-order predicate calculus, *GE Research Lab. technical report 84CRD322*, 1985.
5) 井田哲雄 編：『新しいプログラミング・パラダイム』，第8章　項書換えシステムと完備化手続き（大須賀昭彦），pp. 165–186，共立出版，1989.
6) 坂井　公：Knuth-Bendix の完備化手続きとその応用，コンピュータソフトウェア 4-1，pp. 2–22，1987.
7) 佐々木建昭，古川昭夫：コンピュータ環論，情報処理 27-4，pp. 404–413，1986.

（とやま よしひと）

第8章　構成的プログラミング



林　　晋

8.1　構成的プログラミング?

bit 1986年4月号に，D. Knuth による「算法的思考と数学的思考」(一松 信訳) という面白い論文が載っている．この論文は，算法的思考 (アルゴリズム的思考)，すなわち，プログラマや計算機科学者が行なう思考と，数学者が行なう数学的思考を比較し，その違いを明らかにしようとしたものである．数学的思考の性格を，なるべく客観的にとらえるため，D. Knuth は8冊の数学書と，D. Knuth 自身による “Fundamental Algorithms” の100ページ目の内容について，数学処理，算法などの9項目に対し “得点” を与えて，その思考法を特徴づけ，それを D. Knuth の考える算法的思考の特性と比較している．この9冊の本のうち，100ページ目が最も算法的思考に近かったのは，構成的数学の教科書，E. Bishop : *The Foundations of Constructive Analysis* であった．この構成的数学を使ってプログラミングと，その検証をしてしまおうというのが本稿の話題の**構成的プログラミング**である†.

構成的プログラミングの可能性は，70年代に，R. L. Constable，後藤滋樹，佐藤雅彦らによって指摘され，80年代になって Martin-Löf らによってプログラミング言語における型理論との関連が指摘されて以来，計算機科学者の注目を浴びるようになった．構成的数学は，今世紀の初頭に L. E. J. Brouwer というオ

† 「構成的プログラミング」は，東北大学・佐藤雅彦氏の造語 constructive programming の訳である．

ランダの数学者が始めた直観主義という数学思想に源を持つ数学であるが，計算機科学者にとっての構成的数学は，プログラミング言語と，プログラム検証用の言語が渾然一体となった関数型言語の一種にすぎない．そこで，そのような言語のうちで最も原始的な型付きラムダ計算と呼ばれている高階関数型言語の説明をして，それを使って構成的数学と，そのプログラミングへの応用を説明しよう．

8.2　型付きラムダ計算

というわけで，まず，**型付きラムダ計算**[1)] の説明をする．どのようなプログラム言語でも，二つの基本的要素を持っている．すなわち，データとアルゴリズムである．プログラミング言語の一番基本的なデータは，プログラミング言語の用途によって異なるが，基本的データから複合的なデータを構成する方法や，アルゴリズムの記述法は，どの関数型言語でもさほど変わらない．型付きラムダ計算は，型付きの副作用のない関数型言語の普遍的な機能のみを持つ言語である．ただし，再帰呼出しは，別扱いにするので含まれていない．なぜ別扱いにするのかは後で説明することにして話を続けよう．

複合的データのうち，最も単純なものは，複数のデータをまとめて一つのものと考えたものであろう．Lisp の *cons* セルは，その代表的な例である．型付きラムダ計算も，*cons* セルのような対を持つ．すなわち，a と b が，データならば，その対 $\langle a, b\rangle$ もデータである†．対 $\langle a, b\rangle$ は，Lisp でいえば *cons* セル $(a.b)$ である．Lisp の関数 *car*，*cdr* に対応するものは，*fst*，*snd* と書く．(*fst*，*snd* は，firts, second の略である．）すなわち，

(A)　$fst(\langle a, b\rangle)=a, snd(\langle a, b\rangle)=b$

である．

名前のとおり，型付きラムダ計算は型付きの言語なので，どのようなデータも型を持つ．a, b がデータ型 A, B を持つとき対 $\langle a, b\rangle$ のデータ型は，A と B の**直積型**と呼ばれ $A\times B$ と書かれる．また，a の型が $A\times B$ ならば，$fst(a)$，$snd(a)$ の型は，それぞれ，A, B となる．

---

† 型付きラムダ計算に，対を含めない人も多いが，対があるほうが自然である．

高階言語である型付きラムダ計算では，関数もデータの一種として取り扱う．型Aの値aに，プログラムeの値を対応させる関数を，$\lambda a : A.e$ と書き，**ラムダ式**と呼ぶ．ラムダ式 $\lambda a : A.e$ は，Lisp のラムダ式

(lambda (a) *e*)

に対応する．

関数もデータであるから型を持つ．型Aのデータに，型Bのデータを対応させる関数の型を $A \to B$ と書く．aの型がAであるとき，eの型がBならば，$\lambda a : A.e$ の型は，$A \to B$ である．また，fが型 $A \to B$ を持てば，型Aのデータaにfを適用した $f(a)$ の型はBである．また，$\lambda a : A.e$ は，aから値eを計算するのだから，bに適用すると，その値はeのaをbで置き換えた $e[b/a]$ の値に等しい．すなわち，

(B)　$(\lambda a : A.e)(b) = e[b/a]$

となる．

たとえば，*cons* セルの順番をひっくり返す Lisp の関数

$(lambda(x)(cons(cdr x)(car x)))$

は，型付きラムダ計算では，

$\lambda x : A \times B. \langle snd(x), fst(x) \rangle$

となる．この関数を *reverse* と名づけよう．*reverse* は次のように $\langle a, b \rangle$ の a, b をひっくり返した $\langle b, a \rangle$ を計算する．

$reverse(\langle a, b \rangle) = (\lambda x : A \times B. \langle snd(x), fst(x) \rangle)(\langle a, b \rangle)$　　(B)より

(*)　　$= \langle snd(\langle a, b \rangle), fst(\langle a, b \rangle) \rangle$　　(A)より

$= \langle b, a \rangle$

型付きラムダ計算の定義を，もう少し正確にいうと次のようになる．

型付きラムダ計算のデータと型

D1.　データ a, b の型が A, B ならば，その対 $\langle a, b \rangle$ は型 $A \times B$ のデータである．

D2.　型Aの値から型Bの値を計算する関数は型 $A \to B$ のデータである．

データ型Aの値を計算するプログラムは，型Aを持つということにすると，型付きラムダ計算のプログラムと，その型は次のようになる．

型付きラムダ計算のプログラムとその型

P 0.　変数には，型がちょうど一つ割り当てられている．それを**変数の型**という．変数 a の型が A ならば，a は型 A のプログラムである．

P 1.　$e1, e2$ が型 A, B のプログラムならば，対 $\langle e1, e2\rangle$ は型 $A\times B$ のプログラムである．

P 2.　e が型 $A\times B$ のプログラムならば，$fst(e), snd(e)$ は，それぞれ，型 A, B のプログラムである．

P 3.　a が型 A の変数であるとき，e が型 B のプログラムならば，$\lambda a : A.e$ は，型 $A\rightarrow B$ のプログラムである．

P 4,　f が型 $A\rightarrow B$ のプログラム，e が型 A のプログラムならば，$f(e)$ は型 B のプログラムである．

ラムダ式の型を決めるP 3で a が型 A の変数であることを仮定した．これは，ラムダ式 $\lambda a : A.e$ を作るには，a は必ず型 A の変数でなければいけないということである．これら型の規則は，次のような良い性質を持っている．

性質 1. 型付きラムダ計算の任意のプログラムは，ちょうど一つの型のプログラムとなる．また，その型は規則P 0－P 4により計算できる．

型付きラムダ計算のプログラムは，次の書換えにより実行される．

プログラムの実行方法

(S 1)　$fst(\langle a, b\rangle) \mapsto a,\ snd(\langle a, b\rangle) \mapsto b,$

(S 2)　$(\lambda a : A.e)(b) = e[b/a].$

すなわち，プログラムのある部分がS 1，S 2の書換え規則の左辺にマッチするとき，それを右辺で書き換える．この書換えは，プログラムのどの部分で行なってもよい．書換えが可能な部分は一つとは限らないので，一つのプログラムでもいくつかの異なった方法で実行できる．たとえば，

$\langle fst(\langle a, b\rangle), snd(\langle c, d\rangle)\rangle$

は，$fst(\langle a, b\rangle)$ と $snd(\langle c, d\rangle)$ という二つの書換え可能な部分があるので，

$\langle fst(\langle a, b\rangle), snd(\langle c, d\rangle)\rangle \mapsto \langle a, snd(\langle c, d\rangle)\rangle \mapsto \langle a, d\rangle$

$\langle fst(\langle a, b\rangle), snd(\langle c, d\rangle)\rangle \mapsto \langle fst(\langle a, b\rangle), d\rangle \mapsto \langle a, d\rangle$

のように2通りの実行ができる．また，書換えができなくなったとき，プログラムの実行は終了したといい，そのときのプログラムを実行の結果という．たとえば，$\langle a,b\rangle$ は，これ以上書き換えられないので，上記の二つの書換えの列は $\langle fst(\langle a,b\rangle),\ snd(\langle c,d\rangle)\rangle$ の実行であり，その結果は両方とも $\langle a,d\rangle$ である．また，上記の二つ以外には，実行の方法はないので，このプログラムの実行は必ず停止する．実は，型付きラムダ計算のプログラムは，必ずこれらの性質を持っている．すなわち，次のような性質が成り立つ．

性質 2. プログラムの実行は，実行の方法によらず必ず終了する．

性質 3. プログラムの実行結果は，実行の方法によらない．すなわち，$e1$，$e2$ が，プログムラ e の二つの異なる実行の結果であっても $e1$ と $e2$ は等しい．

型付きラムダ計算に再帰呼出しを含めなかったのは，再帰呼出しを含めると性質2が成り立たなくなるからである．性質2は，プログラムは暴走しないということだから大変良い性質である．しかし，再帰呼出しがないと型付きラムダ計算は，計算機言語としては無力に等しいので，性質2を保ちながら必要な再帰呼出しを追加しなければならない．これについては，後で説明する．

8・3 論理としての型付きラムダ計算

型付きラムダ計算は，一種のプログラミング言語であるが，同時に論理の体系と見なすこともできる．プログラムの型の規則 P1 の「プログラム」を「証明」，「型」を「論理式」，$A\times B$ を A and B と読み替えてみよう．すると P1 は，

P1′. $e1, e2$ が論理式 A, B の証明ならば，$\langle e1, e2\rangle$ は論理式 A and B の証明である．

A and B を証明することは，A と B の両方を証明することだから，A と B の証明の対は，A and B の証明を与えると解釈できる．つまり，P1′ は正しい論理の規則になっている．同じことを P2 でやってみよう．

P2′. e が論理式 A and B の証明ならば $fst(e)$, $snd(e)$ は，それぞれ，論理

式 A, B の証明である．

A and B が証明できているならば，AもBも証明できるからeは，AとBの両方の証明を与える．だから，$fst(e)$ が

A and B だからAである．

という推論によってeからできるAの証明を表わしていると考えれば，これも正しい論理の規則になる．

こんどは，P3, P4を読み替えてみよう．型，プログラムの読替えはP1, P2のときと同じにして，「変数」，$A \rightarrow B$ を，それぞれ「仮定のラベル」，A implies B と読み替えると，次のようになる．

P3′.　aが論理式Aの仮定のラベルであるとき，eが論理式Bの証明ならば，$\lambda a : A.e$ は，論理式Aの証明である．

P4′.　fが，論理式 A implies B の証明，eが論理式Aの証明ならば，$f(e)$ は論理式Bの証明である．

仮定のラベルというのは，

> （a）　A
>
> と仮定せよ．…よってBである．仮定（a）よりBが証明されたので，A implies B である．

というように仮定を後で参照するための名前である．P3′はAという仮定のもとでBが証明できるとき，A implies B が証明できることを表わしている．A implies B の証明であるラムダ式 $\lambda a : A.e$ の $a : A$ は，証明eでaというラベルの付いた A という仮定が使われたことを表わしている．P4′は，A implies B とAが証明できるときには，B が証明できるということなので，P3′もP4′も正しい論理の規則である．このようにP1′-P4′は，簡単な読替えにより論理式の証明の規則になってしまう．このように，プログラムの型の規則が証明の規則に読み替えられることを**Curry・Howardの原理**という．また，プログラムの型の規則としての意味を持つ推論や証明を構成的推論，構成的証明という．たとえば，P1′-P4′は構成的推論である．そして，**構成的数学**とは，構成的推論

のみを使って推論を行なう数学のことである.

8.4 構成的プログラミングとは

構成的プログラミングとは，この Curry・Howard の原理を応用したプログラムの形式的開発の一方法である†. Curry・Howard の原理を限量子 *forall* と *exists* に拡張して，構成的プログラミングの基本原理を説明しよう.

forall $x : A.B$ は，A 型の任意のデータ x に対して論理式 B が成り立つという論理式であるが，これに対応する型を**依存積** (dependent product) と呼び，$\prod x : A.B$ と書く．$\prod x : A.B$ は，入力 x の型が A のとき，出力が型 B であるような関数の型である．単なる関数の型 $A \rightarrow B$ とは異なり，B に x が変数として現われると出力の型 B は入力 x に依存する．たとえば，自然数の型を N として，型 N の n に n 次元零ベクトルを対応させる関数 *zero*,

$$zero(n) = (\underbrace{0, 0, \cdots, 0}_{n\text{個}}),$$

の型は，n 次元ベクトルの型を $vector(n)$ とすると，

$$\prod x : N. vector(n)$$

となる.

依存積の型規則は，次のようになる.

P5. x が型 A の変数であるとき，e が 型 B のプログラムならば，$\lambda x : A.e$ は型 $\prod x : A.B$ のプログラムである.

P6. $e1$ が型 $\prod x : A.B$ のプログラム，$e2$ が 型 A のプログラムならば，$e1(e2)$ は型 $B[e2/x]$ のプログラムである.

P6 の $B[e1/x]$ というのは，B の x に $e1$ を 代入して できる型のことである．たとえば，$zero(3)$ は，$(0, 0, 0)$ であり，その型は $vector(n)[3/n]$，すなわち，$vector(3)$ である．P5, P6 に対応する 証明の規則を，証明への言及を省略した推論規則として書くと次のようになる.

---

† プログラムの形式的開発 (formal development of programs)，形式的に記述された仕様を満たすプログラムを，その妥当性を検証しながら開発する手法．妥当性とは，バグがなく仕様を満たすことをいう.

P5′.　xが型Aの変数であるとき，論理式Bが証明できれば，論理式 *forall* $x:A.B$ は証明できる．

P6′.　論理式 *forall* $x:A.B$ が証明でき，$e2$ が型Aのプログラムならば，論理式 $B[e2/x]$ は証明できる．

P5′ では，xを変数と読み，P6′ では $e2$ をプログラムと読んだ点に注意してほしい．

exists $x:A.B$ は論理式Bを満たす型Aのデータxが存在するという論理式である．これに対応する型は**依存和**（dependent sum）と呼ばれ，$\sum x:A.B$ と書く．依存積 $\sum x:A.B$ のデータは，型Aのデータxと型Bのデータyの対 $\langle x, y\rangle$ である．たとえば，nとn次元零ベクトルの対 $\langle n, zero(n)\rangle$ は，型

$$\sum x:N.vector(n)$$

のデータである．

依存和の型規則は，次のようになる．

P7.　$e1$ が型Aのプログラム，$e2$ が型 $B[e1/x]$ のプログラムならば，$\langle e1, e2\rangle$ は型 $\sum x:A.B$ のプログラムである．

P8.　aが型Bの変数であるとき，$e2$ が型Cのプログラムであり，さらに，$e1$ が型 $\sum x:A.B$ のプログラムであれば，

$$((\lambda x:A.\lambda a:B.e2)(fst(e1)))(snd(e1))$$

は型Cのプログラムである．

対応する推論規則は，次のようになる．

P7′.　$e1$ が型Aのプログラムで，論理式 $B[e1/x]$ が証明できれば，論理式 *exists* $x:A.B$ は証明できる．

P8′.　論理式Bを仮定したとき論理式Cが証明できて，さらに，論理式 *exists* $x:A.B$ が証明できれば，論理式Cは証明できる．

たとえば，

（1）　$\prod n:N.\sum v:vector(n).A$

のデータは，入力nに対して，n次元ベクトルとAのデータの対を出力する関数である．これを論理式と考えれば，

(2) $forall\ n : N.exists\ v : vector(n).A$

となる．すなわち，「すべての自然数nに対し，n次元ベクトルvが存在しAが成り立つ」となる．

そこで，Aが「vはn次元零ベクトル」という式であるとしてみよう．すなわち，Aを $forall\ u : vector(n).(u+v=u)$ とする．こうすると，(2) は

(3) $forall\ n : N.exists\ v : vector(n).$

$$forall\ u : vector(n).(u+v=u)$$

となる．すなわち，「すべての自然数nに対し，n次元零ベクトルvが存在する」という論理式になる．

今度は，これを型として読んでみよう．それには，等式 $u+v=u$ を型として解釈しなければならない．等式$a=b$に対応する型 $Eq(a,b)$ は，

$$Eq(a,b)=\begin{cases}\{true\} & \text{if } a=b\\ \{\ \} & \text{if } a\neq b\end{cases}$$

という型である．すなわち，$Ep(a,b)$ は$a=b$が正しければ $true$ という唯一のデータを持ち，$a=b$ が正しくなければデータを持たない空な型になる．(3) の型としての読替えは，

(4) $\prod n : N.\sum v : vector(n).\prod u : vector(n).Eq(u+v,u)$

となる．

この型のデータが，どのようなものになっているか考えてみよう．これは，依存積であるから，そのデータは関数である．fを，この型の関数であるとしよう．nが自然数であれば，$f(n)$ はn次元ベクトル v と型 $\prod u : vector(n).Eq(u+v,u)$ のデータ gの対 $\langle v,g\rangle$ である．uをn次元ベクトルとすると，$g(u)$ は $Eq(u+v,u)$ のデータになる．したがって，$Eq(u+v,u)$ は空ではないから，Eq の定義より $u+v=u$ が成り立つ．これは，任意のuに対して成り立つのだから，vは零ベクトルである．すなわち，型 (4) のデータは，nからn次元ベクトルvと，vが零ベクトルであることを示す関数gの対を計算する関数である．

論理式(3)は，零ベクトルの存在定理であったが，それが構成的推論で証明されたとしよう．その証明eは (4) の型を持つプログラムとみなせるので，$fst(e)$ は，n次元零ベクトルとなる．すなわち，零ベクトルの存在が構成的に証明できれば，零ベクトルを計算するプログラムが得られる．また，このよう

にして作られたプログラムは，それ自身が証明であるから妥当なプログラム (verified program) である†．つまり，プログラムの仕様は，プログラムの出力の値の存在を表わす論理式に対応し，プログラムと，その検証は条件を満たす値の存在証明に対応する．すなわち，構成的プログラミングとプログラムの形式的開発の間に次のような対応がある．

構成的数学	プログラムの形式的開発
論理式	型
証明	プログラムと検証
存在定理	仕様
存在定理の証明	仕様にあったプログラムの作成と，その検証

この対応を利用してプログラムの形式的開発を行なうのが構成的プログラミングである．

実際の構成的プログラミングのシステムとしては，Göteborg 大学のシステム (名前はついていない)[2]，Cornell 大学の Nuprl[3]，筆者の PX[4] などがある．Göteborg 大学のシステムと Nuprl は，Curry・Howard の原理に基づいてプログラムを構成するが，PX は realizability interpretation という Curry・Howard の原理の変種によってプログラムを構成する．以下では，PX での例を使って，どのように構成的プログラミングを行なうかを説明しよう．

PX は Stanford 大学の論理学者 S. Feferman の形式系 T_0 を構成的プログラミングのために設計しなおした形式系に基づくシステムである．PX の理論は Lisp のプログラムについての理論になっていて，PX での構成的プログラミングによって作られるプログラムは Franz Lisp のプログラムである．

8.5　構成的プログラミングにおける再帰呼出し

型付きラムダ計算には再帰呼出しがなかったので，それだけでは役に立つプログラムを書くことはできない．Π や Σ を使っても事情は同じである．そこで構成的プログラミングにより役に立つプログラムを書くには再帰呼出しに対応

† 妥当なプログラム＝妥当性の証明されたプログラム，205ページの脚注参照．

する推論規則が必要となる．再帰呼出しに対応する推論法則は数学的帰納法のような帰納法の原理である．

たとえば，自然数 a を，0でない自然数 b で割った商と余りを計算するプログラムを構成する問題を考えよう．このプログラムの仕様は

$$\text{(A)}\quad \textit{forall } a : N.\, \textit{forall } b : N^+.\ \ \textit{exists } q : N.\, \textit{exists } r : N.\ \ (a = b*q + r \text{ and } r < b)$$

となる．N^+ は，0でない自然数の型である．これを，a についての数学的帰納法で証明してプログラムを構成する．PX の数学的帰納法は次のような推論である．

$$\frac{\text{(MI1)}\quad \mathrm{P}(0) \qquad \text{(MI2)}\quad a : N^+ \text{ かつ } \mathrm{P}(a-1) \text{ ならば } \mathrm{P}(a)}{\textit{forall } a : N.P(a)}$$

（A）を証明するには，

$$\mathrm{P}(a) = \textit{exists } q : N.\textit{exists } r : N.(a = b*q+r \text{ and } r < b)$$

とする．すなわち，

$$\text{(B1)}\quad \textit{exists } q : N.\textit{exists } r : N.\ (0 = b*q+r \text{ and } r < b)$$

$$\text{(B2)}\quad a : N^+ \text{ かつ } \textit{exists } q : N.\textit{exists } r : N.\ (a-1 = b*q+r \text{ and } r < b) \text{ ならば } \textit{exists } q : N.\textit{exists } r : N.\ (a = b*q+r \text{ and } r < b)$$

が証明できればよい．(B1) では，$q=r=0$ とすればよい．(B2) では，$a-1=b*q'+r' \text{and } r'<b$ と仮定して，$r'+1=b$ と，そうでないときの場合分けをする．$r'+1=b$ のときには，$q=q'+1, r=0$ とする．$r'+1\neq b$ のときには，$q=q', r=r'+1$ とする．こうすると $a=b*q+r \text{ and } r<b$ が成り立つので，証明規則 P7′ と P8′ と場合分けによる推論を使って (B2) が証明できる．この証明から PX は，次のようなプログラムを構成する．

```
(defun f (a b)
  (cond
```

```
        ((equal 0 a) (list 0 0))
        (t (let (((q r) (f (sub1 a) b)))
             (cond ((equal (add1 r) b) (list (add1 q) 0))
                   (t (list q (add1 r))))))))
```

ただし，(let (((q r) (f (sub1 a) b))) …) は，(f (sub1 a) b) の値が (q r) にマッチしたとき，そのマッチングのもとで，…を計算し，その値を返す．また，add1 は，1を加える関数である．(f a b) は a の b による商 q と余り r のリスト (q r) を計算する．f の定義で，(f (sub1 a) b) という f の再帰呼出しが使われていることに注意してほしい．PX で変数 a1 について数学的帰納法が使われると新しい関数名が生成され，それを f とすると，

```
(defun f (a1 … an)
  (cod ((equal a1 0) …)
       (t … (f (sub1 a1) … an) … )))
```

という形のプログラムが作られる．この形の再帰呼出しは，論理学で原始帰納法†と呼ばれているもので，a1 は自然数だから，f の再帰呼出しごとに，a1 の値が小さくなるので，この再帰呼出しのせいで計算が無限ループに入ることはない．したがって，型付きラムダ計算にこの形の再帰呼出しを追加しても，プログラムは必ず停止する．また，数学的帰納法を使った構成的プログラミングでは，かなり広い範囲の計算可能関数がプログラムできることが知られている．実際，NP-完全なアルゴリズムも数学的帰納法によってプログラムできる．しかし，十分広い範囲の関数がプログラムできるといっても，効率の良いプログラムができるというわけではない．効率の良いプログラムを書くには数学的帰納法だけでは不十分である．たとえば，上のプログラム f は，たとえ，a＝b であって，商が1であっても f が a 回再帰的に呼び出されてしまう．これでは困るので，PX では CIG 帰納法という帰納法のクラスが定義されていて，効率の良いプログラムを書くことができるようになっている．それでは，CIG 帰納法を使って仕様（A）を満たす，もっと効率の良いプログラムを構成して

---

† 原始帰納法 (primitive recursion). 本稿では，recursion と induction を区別するために，「再帰呼出し」と「帰納法」という言葉を使った．だから，本当は「原始再帰法」とでも呼んだほうがよいのだが，この言葉に限って慣例に従うことにする．

みよう．

CIG 帰納法を使って（A）を証明する方法はいろいろ考えられる．ここでは，最も効率の良いものではないがわかりやすい例を紹介しよう．a の b による商と余りを計算するには，b で a を繰り返し引いていって a が b より小さくなったら a を余りとし，引いた回数を商とすればよい．このアルゴリズムを構成してみよう．

1. 0 でない自然数 b に対して，$D(b)$ という補助的な型（集合）を定義する．$D(b)$ は，N の部分集合で

(1.1)　$a<b$ ならば $a : D(b)$,

(1.2)　$a<b$ でなく $a-b : D(b)$ ならば，$a : D(b)$,

と帰納的に定義する．

2. $D(b)$ の定義より，a を自然数とすると，

$$\frac{\begin{array}{ll}(2.1) & a<b \text{ ならば } \mathrm{P}(a), \\ (2.2) & a<b \text{ でなく } \mathrm{P}(a-b) \text{ ならば，} \mathrm{P}(a)\end{array}}{\mathit{forall}\ a : D(b).\, \mathrm{P}(a)}$$

という帰納法が正しい推論法則となる．これを $D(b)$ の **CIG 帰納法**という．$\mathrm{P}(a)$ を $\mathit{exists}\ q : N.\ \mathit{exists}\ r : N.\ (a=b*q+r \text{ and } r<b)$ として，この CIG 帰納法を使って $\mathit{forall}\ a : D(b).\, \mathrm{P}(a)$ を証明しよう．(2.1) は $q=0$, $r=a$ とすれば成り立つ．(2.2) は，$a-b=b*q'+r'$ and $r'<b$ として $q=q'+1$, $r=r'$ とすれば成り立つ．すなわち，

(C)　$\mathit{forall}\ a : D(b).$

$\mathit{exists}\ q : N.\ \mathit{exists}\ r : N.$

$(a=b*q+r \text{ and } r<b)$

が証明された．

3. $a : N$, $b+N^{+}$ のとき，$a : D(b)$ であることを，a についての数学的帰納法で証明する．

4. （C）と 3. より（A）が証明される．

この証明から構成されるプログラムは，

```
(defun g (a b)
  (cond
```

```
((less a b) (list 0 a))
(t (let (((a1 a2) (g (sub a b) b)))
          (list (add1 a1) a2)))))
```

となる.

このプログラムでは, gは商の回数しか呼び出されないのでfよりはるかに効率が良い. もちろん, このプログラムも再帰呼出しを使うので, あまり効率は良くないが, この形の再帰呼出しは, カウンタを使えば, 簡単なプログラム変換により while などの繰返しにかえることができることが知られている.

実は, 正規プログラムについての Hoare 論理でのプログラムの形式的開発[5)]は, PX の構成的プログラミングに自動的に変換できる. また, そのような変換によって得られたプログラムの再帰呼出しは, 必ず末尾再帰呼出しになる[†1]. この意味で, PX の構成的プログラミングは, Hoare 論理によるプログラムの形式的開発の拡張になっている.

構成的プログラミングにおける帰納法の役割を明らかにするために, f の証明とgの証明を比較してみよう. fの証明では, 数学的帰納法が一度だけ使われている. 一方, g の証明では, ステップ2と3で二つの帰納法が使われている. そして, f では数学的帰納法から原子帰納法が構成されている. 他方, g ではステップ2の CIG 帰納法からgの再帰呼出しが構成され, ステップ3の数学的帰納法に対応するものは, どこにも使われていない. これは, PX がステップ3はプログラムの構成に関係がないことを検出するからである. 実は, ステップ2の (C) の証明からプログラムを構成してもgが得られる. $D(b)$ は, aがbより小さくない限りbで引き続けるというプロセスが停止するaの領域を定義していて, ステップ3は, すべての自然数が $D(b)$ に属していることを証明している. $D(b)$ にaが属せばプログラム (g a b) は停止するから, ステップ3はgの停止性の証明であると考えられる. また, ステップ2はgの部分妥当性の証明と考えられる[†2].

---

†1 末尾再帰呼出し (tail recursion). 簡単に繰返しに変換できる再帰呼出しの一形態. たいていの Lisp コンパイラは, この変換を行なう. また, スタックを消費しないで末尾再帰呼出しを実行するインタプリタもある.

†2 部分妥当性 (partial correctness). プログラムが停止すれば, その出力は仕様を満たすこと. ここでは, それより少し広い意味で使われている.

伝統的なプログラムの形式的開発では，部分妥当性は，プログラムの定義から定義される計算帰納法（computational induction）を使って証明され，停止性は数学的帰納法などの論理学や数学で使われる帰納法によって証明される．fの証明は，この二つの帰納法が，偶然同一であったに過ぎない．構成的プログラミングでは，この二つの帰納法を同一にするプログラミングが好まれるが，効率の良いプログラムを構成するためには，二つに分けたほうがよいことが多い．また，上に注意したように停止性の証明はプログラムに影響を及ぼさないので，この証明をサボッたり構成的数学ではなくふつうの数学を使って証明してもかまわない．この方法は，PXではじめて導入された方法で，完全な構成的な証明というプログラミングよりはるかに面倒な作業を必要とする従来の構成的プログラミングの方法を著しく簡単にしている．

8.6 おわりに：構成的プログラミングに未来はあるか？

構成的プログラミングは，未完成な方法で，そのいく末がどうなるのかは筆者にもわからない．現在のところ，小規模ながらプログラムと呼ぶに値するものを構成でき，完全な検証を行なえるのはNuprlとPXのみであるようだし，この二つのシステムでのプログラム開発の例も多くはない†．アイデアが出てから15年以上たってもこんな状態だから，当然のことながら批判も少なくない．

多く寄せられる批判は，プログラミングより難しい数学を使ってプログラムをしてもしようがないというものである．これは，構成的プログラミングがプログラムの形式的開発の一方法であり，プログラム検証の問題を含んでいることを忘れた批判である．もちろん，このような批判は，定理証明の問題を含むプログラムの形式的開発は実際には不可能であるという批判であることも多い．しかし，最近ではBoyer・Mooreの定理証明系によって，Gödelの不完全性定理が証明されたように，かなり複雑な証明も計算機でチェックできるようになってきている．

---

† PXで構成された一番大きなプログラムは，命題論理の定理証明プログラム（式が正しければ証明を，正しくなければ反例を，計算するプログラム）である．

また，筆者はPXにおけるプログラム開発の経験から，完全な検証は不可能でも，部分的な検証や，検証とテストの組合せは十分可能であり，ソフトウェアの信頼性を飛躍的に向上させるために有効であると確信している．また，プログラムの情報を妥当性の問題以外に利用できる可能性もある．たとえば，萩谷昌己氏[6)]は構成的型理論を用いたプログラムの一般化の方法を提唱している．構成的命題論理を応用したプログラム合成系PRIZ[7)]でCADシステムが開発されたという話も聞いた[†1]．PRIZでは本稿で説明したような完全な仕様を記述し検証することはできないが，プログラムは，ほぼ自動的に合成される[†2]．構成的プログラミングの技法は地味ながら着実に進歩し続けているのである．

最後に参考文献をあげておこう．Curry・Howardの原理については，1）に解説がある．PXについて詳しく知りたい人は4）を読んでいただければ幸いである．Nuprlについて知りたい人には3）を勧める．

参考文献

1) Hindley, J.R. & Seldin, J.P.: *Introduction to Combinators and lambda-calculus*, Cambridge University Press, 1986.
2) Nordsrtöm, B. & Petersson, K.: Types and Specifications, in *Proceedings of IFIP '83*, 1983.
3) Constable, R. et al.: *Implementing Mathematics with the Nuprl proof Development System*, Prentice-Hall, 1986.
4) Hayashi, S. & Nakano, H.: *PX: A Computational Logic*, The MIT Press, 1988.
5) Alagić, S. & Arbib, M.A.: *The Design of Well-Structured and Correct Programs*, Springer-Verlag, 1978.
6) Hagiya, M.: Generalization from partial Parameterization in Higher Order Type Theory, preprint.
7) Tyugu, E.: *Knowledge-based Programming*, Addison-Wesley, 1988.

---

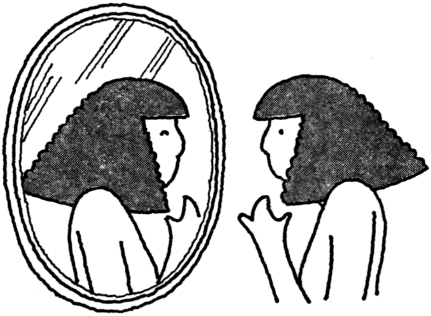
†1 PRIZの理論的基礎を研究しているソ連の論理学者G. Mintsから聞いた話である．

†2 NuprlやPXのようなシステムとPRIZのようなシステムの統合は面白い問題であろう．

（はやし すすむ）

第9章

メタプログラミングとリフレクション



田中二郎

9.1 はじめに

ちょっと前になるが，「究極の…」という言葉がはやっていた．すべて物事に凝る性質の私としてはこの言葉が大好きである．この言葉は，「究極の××を求めて」だとか，「これぞ究極の××」という具合に使われて，求め得る限り最良の品質を持つもの（でなかなか容易に手に入らないもの）を指す場合が多かった．

そこでここでは，「究極」のプログラミング言語について考えてみたい．過去から現在に至るまでコンピュータのためにさまざまなプログラミング言語が提案され，それらはさまざまに使われてきた．このような中で「究極」のプログラミング言語とは一体どのようなものであろうか．

むろん，この問にはさまざまな解答が可能である．ある人はプログラミング言語の「記述力」を挙げ，ある人はプログラミング言語としての「簡潔性」を挙げるかもしれない．

しかしながらよく考えてみるとプログラミング言語において「記述力」と「簡潔性」とは相反する概念である．すなわち一般にはプログラミング言語の「記述力」が増せば増すほど，言語全体としては複雑でわかりにくいものになっていく．そこで考えるに何か「簡潔性」を保ちつつ，言語の「記述力」を高めるための方法はないものだろうか．実際のところ，そのような方法としてすでにいくつかの枠組みが知られている．その中でもメタプログラミング機能や

リフレクション機能は特に最近話題となっている枠組みである．

9.2 計算システム

メタプログラミングやリフレクション機能について説明する前に，文献 1)，2) を参考に，まず，計算システム一般について説明する．計算システムとは，コンピュータなどの計算のしくみを抽象化したものと考えればよい．一般の計算システムを図に示したのが**図 9.1** である．すなわち計算システムは「プログラム」と「データ」と「実行器」の三つからなる．計算システムは問題領域における現象や動作をモデル化し，それをプログラムとデータの形にして格納する．プログラムは問題領域に対しての「操作」をモデル化し，データ部分では問題領域の「構造」をモデル化する．実行器はこれらプログラムとデータを実行する計算のエンジンすなわちコンピュータでいえば CPU にあたるものである．

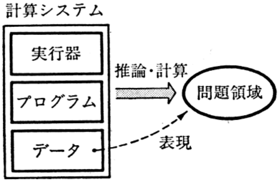


図 9.1　計算システム

9.2.1 因果結合システム

因果結合システム（causally connected system）とは，計算システムの特別な場合であり，「データ」と「問題領域」とが因果的につながっている（causally connected）ような計算システムである．

こういった因果結合システムの一例としてロボットの制御などのリアルタイムシステムが考えられる．リアルタイムシステムでは，問題領域における変化が計算システムのデータとして伝わり，計算システムにおけるデータの変化はロボットの動作として問題領域に反映される．すなわち「データ」と「問題領

域」が因果的に結合（causally connected）されている．

このような因果結合をどのように実現するかということが次の問題であるが，一般に因果結合の実現手段としてはさまざまな可能性が考えられる．上に挙げたリアルタイムシステムなどではセンサを外部世界に置き，センサからの情報を次々にデータに取り込めばよい．また，計算システムにおける内部データの変化は，外の世界に対しての行動，たとえばロボットのモータへの制御情報として次々に出力されるようにしてやればよい．

もう一つの可能性としては，問題領域そのものをデータとするということが考えられる．この場合は「問題領域」と「データ」とが因果結合されているというより，同じものが「問題領域」と「データ」の２通りに使われるといったほうが正確であるが，こうした扱いは，「問題領域」が計算システムと同じ種類の構造（たとえば Lisp におけるＳ式）で記述されているとき可能となる．

9.2.2 メタシステム

次にメタシステムであるが，メタシステムとは，その計算システムの問題領域がまた計算システムであるようなシステムである．それを図に示したのが図 9.2 である．メタシステムのデータは，問題領域であるほかの計算システム（オブジェクトシステム）をモデル化したものになっている．またメタシステムのプログラムは「メタプログラム」と呼ばれ，オブジェクトシステムの「操作」をモデル化したものになっている．

なお，メタシステムにおけるモデル化の程度であるが，これはメタシステムにおいてオブジェクトシステムの何をどの程度モデル化したいかにより，さまざまなレベルのモデル化が可能である．

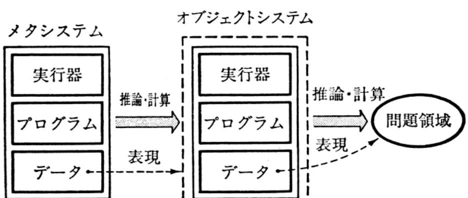


図 9.2　メタシステム

メタプログラムやメタシステムの実際の例であるが，これらについてはすでにさまざまな領域で研究がなされてきている．起源的には，メタプログラムはLispで始められた技法であると思われる．すなわちLispにおいては，プログラム，データともにS式で表わされるので，LispシステムをLispで表記することが可能である．こうした事実は，Lisp 1.5[3]が提案されたときより，Lispの万能関数（universal function）として知られてきた．Lispの万能関数とはS式で書かれたプログラムに引数を与え，その値を計算するためのアルゴリズムをLispで書いた（すなわち自分の働きを自分自身の記述法で書いた）関数のことである．自分の働きを自分自身で表記するという観点からこれらを概念的に整理した仕事としては，G. Steeleらのmeta-circular interpretersが有名である[4]．

一方，メタプログラムは，Prologにおいても，いわゆる“Prolog in Prolog”（すなわちPrologによるPrologの表現）として知られてきた．これについては，以下に示すような，いわゆる4行メタインタプリタというのが有名である．

```
exec(true) :- !.
exec((P, Q)) :- !, exec(P), exec(Q).
exec(P) :- clause((P :- Q)), exec(Q).
exec(P) :- !, P.
```

この4行のプログラムは簡単ではあるが，これだけで“Prolog in Prolog”になっている．このプログラムの意味は以下のとおりである．まず，実行されるべきゴールがtrueであれば成功して終了する．実行されるべきゴールが複数個あれば，それを分解し，一つ一つをそれぞれ実行する．ゴールがtrueでも複数個でもないときは，述語clauseが，与えられたゴールにユニファイ可能な定義節を見つけ，その定義節のボディにゴールを展開する．また，それ以外のときは与えられたゴールはシステム述語であるのでそれを解く．このように，このプログラムはたった4行ではあるがPrologプログラムの実行過程の操作をモデル化したメタプログラムになっている．

こうした“Prolog in Prolog”の考え方をメタシステムという観点から整理し，いわゆる「デモ述語」の提案をしたのはBowenとKowalski[5]である．デモ述語

は“demo (Prog, Goals)”の形で使われ，Prog から Goals が証明可能であるかを示す述語である．デモ述語のインプリメントは簡単で，陽にプログラムとしてProg が指定されていることを除けば，exec のプログラムと同様のメタプログラムで実現できる．

こうしたシステムではこれらの例でも明らかなようにオブジェクトシステムがメタインタプリタの形で実現されている．しかしながら，こういったシステムは，必ずしもメタインタプリタの形で表現されなければならないというわけではない．なんらかの形でメタシステムからオブジェクトシステムを定義できればよいのである．メタインタプリタによらないメタシステムとしては Weyhrauch の FOL[6] などがよく知られている．

9.2.3 リフレクティブシステム

リフレクション（reflection）とは，「自分自身」について感知したり自分自身を変更したりすることである．もしもそのような能力をプログラミング言語やシステムが持つならば，それらは強力なシステムを簡潔に記述することにつながる．たとえば，オペレーティング・システムなどを記述することを考えてみよう．オペレーティング・システムにおいてはメモリや CPU タイムなどのシステムの持つ自己の資源を管理することが要求されるが，このとき，システムが何らかの形で自己の状態や能力などを動的に感知し，それによってそれに応じた行動をとれるとしたら便利である．体力が余っていれば，忙しく仕事をし，忙しくなってきたらあまり新しい仕事は受け付けない．また自己の能力の限界を知っていて，できそうもないと思ったらあきらめてしまうといったようなことも可能となる．

こうした自己感知や自己変更については，高級言語のように「プログラム」と「データ」を分けて考えるのではなく，アセンブリ言語のように，時には「プログラム」を「データ」と解釈したり，自分で自分のプログラムを書き換えたりする能力が（もっと整理された形で）必要となる．

リフレクティブなシステムを図に示したのが**図 9.3** である．リフレクティブシステムとは，オブジェクトシステムとして自分自身を持ち，それが自分自身のデータと因果結合されているようなメタシステムである．

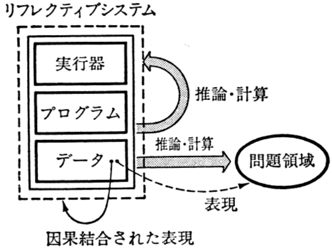


図 9.3　リフレクティブシステム

リフレクションについていい出したのはFOLが最初であるが，具体的にリフレクションのメカニズムを明らかにし，リフレクション機能を実現したのはB. C. Smith の 3-Lisp[7)]である．

9.3　3-Lisp

B. C. Smith は，まず，Lisp 1.5 のようなふつうの Lisp のことを **1-Lisp** と呼んでいる．Lisp ではプログラムもデータもＳ式として実現されているために，メタインタプリタでオブジェクトシステムを記述することが比較的容易である．B. C. Smith は 1-Lisp を検討していくうちに，1-Lisp においては「データの表示するもの (designation)」と「データの評価 (evaluation)」との区別が不明確であることを発見した．こうした不明確さは，メタシステムにおいてプログラムをデータとして扱うときに，混乱を招くと考えた．今，データの表示 (designate) をΦ，データの評価 (evaluate) をΨとするとき，1-Lisp でのΦとΨの例を図 9.4 に示す．

この図でもわかるように，データの表示Φとは，「データ」とその意味するところの「概念」を結び付ける関数である．たとえば，アトムの「3」は「Three」という概念に対応し，Ｓ式 (＋ 1 2) もやはり「Three」という概念に対応している．また「'X」は「X」に対応している．

一方，データの評価ΨはＳ式の変形 (reduction) に関連した概念であり，データはΨにより簡単化される．たとえば，Ｓ式「(＋ 1 2)」はΨで評価するとア

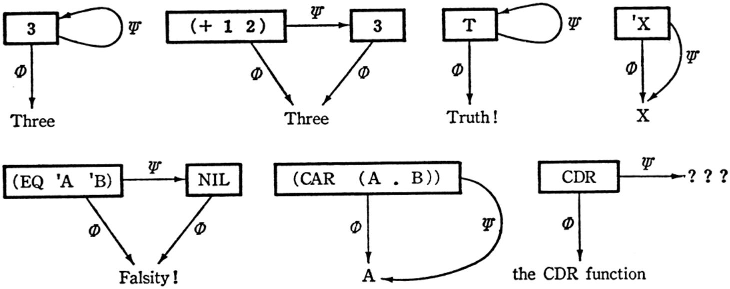


図 9.4 Lisp の表示と評価

トム「3」であり，アトムは評価してもそれ以上値は変わらない．また「'X」はΨで評価するとXになり，さらにXをΨで評価すると，また異なった値となる．

この例でもわかるように，1-Lisp ではΦ(designate) とΨ(evaluate) の意味するところは微妙に異なっている．こうしたΦとΨの複雑さを B.C. Smith は嫌った．本来Φとはメタレベルの「データ」とオブジェクトレベルでの「概念」の対応を意味するはずである．またΨとはメタレベル内のデータの変形であるはずである．したがって，Ψは一度評価したらある一定の値（標準形）となり，さらに評価しても不変であるべきである．

2-Lisp というのは，実はこれらの不明確さを一掃したり折り目正しい Lisp である．2-Lisp では，表現をΨで評価すると標準形 (normal form) になり，これをさらに評価してもこれ以上値は変わらない．たとえば，Ψ'(+ 2 3) ⇒ '5 であり，Ψ('X) ⇒ 'X である．

またΦは 2-Lisp では↑と書き，たとえば↑(+ 2 3) ⇒ '5 である．これはオブジェクトレベルからメタレベルへの対応に相当する．Φの逆関数に対応する操作を↓と書き，たとえば↓('5) ⇒ 5 である．これは逆にメタレベルからオブジェクトレベルへの対応に相当する．

この 2-Lisp にリフレクションのための言語機能を追加したのが 3-Lisp である．すでに前で述べたように，リフレクション機能とは，プログラムを実行しながら自分の現在の実行状態を取り出したり，変更したりする機能のことであ

る．こうした機能の実現のためには，前に図9.3で示したような構成，すなわちオブジェクトシステムとして自分自身を持ち，これが自分自身のデータと因果結合されているような構造を実現してやる必要がある．

そのために，3-Lispでは，メタインタプリタを利用し，自分自身であるオブジェクトシステムがメタインタプリタの管理下で動くようにしている．オブジェクトシステム中で自己状態に関する情報が欲しいときには，それを動かしているメタシステムに問合せをする．そしてメタシステムから情報を得て自己状態を参照あるいは変更し，変更された自分の状態を再びメタシステムに戻してやる必要がある．ここで正確にはメタシステムから情報を得ることをreification（具体化）といい，メタシステムに情報を戻すことを（狭い意味での）reflection（反射）という．

3-Lispではメタな情報としてプログラム全体の継続（continuation）と変数の束縛環境（environment）の二つを考えた．3-Lispのメタインタプリタでは，実行される関数が，変数環境だけでなくプログラムの継続を常に持ち運ぶようになっている．(「継続」とはプログラム実行における「残りの仕事」に対応する特殊な関数である．)

このようなプログラムの継続を常に持ち運ぶようなメタインタプリタのことをcontinuation-passing styleのメタインタプリタと呼ぶ．3-Lispのリフレクティブな関数では，ちょうどLisp 1.5のFEXPR関数が関数呼出しの際に自由に環境を取り出せるように，プログラムから自由に継続や環境を取り出すことができ，それに変更を加えて戻せるようになっている．

たとえば，今ある変数が束縛されているかどうかを調べる関数BOUNDは，以下のように定義できる．

```
(Defun BOUND
    (lambda reflect ((var) env cont))
      (if (bound-in-env var env)
      (cont '$T)
      (cont '$F)))
```

ここでlambda reflectの引数として環境と継続が取り出されていることに注意されたい．こういったリフレクティブな機能を使うと，プログラミング言語はよ

り強力になり，以前はプリミティブな関数と思われていたものまでユーザ定義関数として書けるようになる．たとえば，Lisp の throw/catch（throw を実行するとそれに対応する catch に制御が移る．要するに Lisp における goto 文）だとか lambda, if, quote などをユーザ定義関数として書くことができる．

9.4 リフレクティブタワー

さて，こうしてリフレクション機能は言語に強力な枠組みを提供することになる．つまり，より小さな核（2-Lisp）で強力な言語（3-Lisp）を記述できる．ここで関数がリフレクティブに定義されると，実際にはこの関数は言語の走っているオブジェクトレベルより一段上のメタレベル，すなわちメタインタプリタで解釈実行されることになる．

さて，そこで問題が一つ生じる．3-Lisp ではオブジェクトシステムのメタインタプリタを書いてメタシステムを実現するのだが，そのメタインタプリタを記述するにはどうしたらよいだろうか．2-Lisp で記述できれば何も問題は起きないが，ここで仮定している 2-Lisp は本当に小さな核である．すでに述べたように，throw/catch, lambda, if, quote なども定義されていない．そこで，記述するのに 3-Lisp を使うことにするとどうだろうか．リフレクティブな関数は 1 段上のレベルで解釈されるので，そのメタインタプリタの解釈にもう 1 段上のメタが必要になる．すなわちメタのメタが必要になってしまう．そしてメタのメタの世界においてリフレクティブな関数を実現するには，さらにメタのメタが必要になり，結局のところメタの無限の階層が必要となる．このメタの無限の階層のことを B.C. Smith は**リフレクティブタワー**（reflective tower）と呼んだ．

要するに問題はメタな世界ではリフレクティブな関数を使わなければよいのだが，それを使わない場合，オブジェクトレベルの表現力とメタレベルの表現力に違いが出ることになり，気分がよくない．

次に無限タワーをどのようにインプリメントするかという問題であるが，実際のインプリメントでは，メタな世界が必要になったときに，必要に応じてメタな世界を作ることにしてこの問題を解決している．

9.5　GHCにおけるリフレクション

以上リフレクションと3-Lispについて簡単に説明した．このあたりで，ICOT((財)新世代コンピュータ技術開発機構）や筆者らを中心に研究を進めている並列論理型言語 GHC にリフレクション機能を付加する試みについて紹介したい．

ICOTとは10年計画で第五世代コンピュータを開発する使命をおびた財団である．このための柱として，並列計算機の開発とProlog風の並列論理型言語の使用を二大前提としている．すでに並列論理型言語については GHC (Guarded Horn Clauses)[8] という言語を提案し，すべてのシステムをこの言語に基づいて構築すべく研究を進めている．

したがって，並列計算機用のオペレーティング・システムについても GHC で記述される必要があるが，GHC の枠内に入りきれない機能についてそれぞれをどのように処理するかという問題がある．いわゆるそれら「メタ機能」について，組込み述語として埋め込んだり，副作用として処理したり，いわば場当り的に処理していたのでは，言語の処理系も膨大なものになってしまう．そこで，もっと統一的に見通しよく扱う方法はないかと開始したのがリフレクションの研究の動機である．

9.5.1　並列論理型言語 GHC

GHCは並列論理型言語であり，言語レベルでプロセスを動的に生成でき，プロセス間の同期の記述なども容易である．これらのGHCの特徴から，GHCでオペレーティング・システムを書いてみたり，並列システムのシミュレーションをやってみたらどうかということを思いつく．

しかしながら実際にこれらのことを試みると，意外と大変である．その原因としては，現在の GHC では，オブジェクトレベルとメタレベルの区別が曖昧で，かつメタ機能が不十分であることが挙げられよう．

たとえば，いまGHCを走らせることができるGHCマシンがあると仮定し，そのオペレーティング・システムについて考えよう．オペレーティング・シス

テムはゴールを読み込み，そのタスクを実行し，実行結果を出力できることが必要である．しかしながら，現在の GHC では，オブジェクトレベルとメタレベルの機能の区分が不明確であるため，こうしたオペレーティング・システムが簡潔に記述できない．並列システムのシミュレーションにしても状況は同じである．通常，並列な実体に対応するプロセスを GHC のプロセスで記述するが，そのプロセスから発生するメタレベルの現象を GHC の枠内で扱えない．

こうした GHC の欠点を補うため考えられているのが，GHC へのリフレクション機能の付加である．我々の考え方は 3-Lisp にヒントを得たものであるが，GHC の場合は並列性を持ち，またゴール実行の成功や失敗などのメタレベルの情報がプログラム実行中に発生するので，若干リフレクション実行機能のための機構は複雑になる．

9.5.2 GHC の言語仕様

GHC については，すでに第1章で解説したとおりである．また単行本も出版されている[8]ので，その概要については明らかである．ここでは Prolog との違いだけを簡単に説明する．

まず，GHC の定義節は，

H :- G|B

の形をしていて，ここでGはガードゴールの列，Bはボディゴールの列である．また，|はコミットオペレータと呼ばれ，Prolog における！(カット）を並列化したものである．(ここでは簡単化のために GHC のガード部のゴールには，大小関係や等号など簡単な条件を表わす組込みのゴールしか許さないと仮定する．このようなシステムを特に **Flat GHC システム**と呼んでいる．)

あるゴールPが与えられたとき，それがある定義節のヘッドHと適合(matching) 可能であり，ガードGを満たすならば，Pはボディゴールの列Bにリダクションされる．適合可能かどうかいまだわからない場合，そのゴール実行はゴールにほかから束縛情報が来るまでサスペンドする（これが GHC における同期規則である)．適合可能な節がないとき，そのゴール実行は「失敗」する．ゴールが次々実行されてリダクションされたゴールがすべてなくなれば，ゴール実行は「成功」する．

Prolog と比較して違うところは，すべての定義節がコミットオペレータを持つこと，ゴール列はすべて並列に評価してよいこと，リダクションの過程に情報が来るまで待つという同期の機構が組み込まれていること，また一度リダクションされたゴールは再びバックトラックを起こすことがないこと，などである．

9.5.3　GHC のメタインタプリタ

さて，GHC の一番簡単なメタインタプリタは，Prolog と同様に，以下に示すように記述できる．（これは，Lisp の場合でいうと，1-Lisp のメタインタプリタに相当する．）

```
exec(true) :- true|true.
exec((P, Q)) :- true|exec(P), exec(Q).
exec(P) :- not-sys(P)|reduce(P, Body), exec(Body).
exec(P) :- sys(P)|P
```

9.2.2 項の Prolog のメタインタプリタと比較してみれば明らかなように，このメタインタプリタはPrologのメタインタプリタとほとんど同じである．ただし，よく見ると Prolog の！（カット）の代わりに，GHC では |（コミット）が使われている．また，定義節を見つけるのに Prolog では clause を使っていたが，GHC では述語 reduce が使われている．述語 reduce は，与えられたゴールに適合可能な定義節を見つけ，それらの定義節の中でガードが解ける定義節を一つ探し，その定義節のボディにゴールを展開する．（そして一度展開されたゴールは再びバックトラックすることはない．）

しかし，このメタインタプリタはこのままでは単に GHC プログラムの実行を最も表層のレベルでシミュレートしているだけであり，あまり役に立たない．このメタインタプリタを修正あるいは拡張し，有用な情報をシミュレートできるようにしたい．

そこで，簡単な拡張として，exec を2引数にして exec(G, R) とすることを考える．この exec は，ゴールGを実行し，そのゴールが成功するとRに success，失敗するとRに failure を返す．（Flat GHC システムでは，ガード部には組込みの述語しか許さないので，ボディゴールが失敗すると，GHC プログラ

ム全体が失敗することになる．したがって，このような2引数 exec を考えることにより，exec の中のゴールが失敗しても外の世界全体が失敗するのを防ぐことができる．）ここでは記述は省略するが，このような2引数 exec は，1引数の exec を多少拡張することによって容易に実現できることは明らかであろう．

次の拡張は，exec のゴール実行を外の世界（すなわち，メタシステム）からコントロールできるようにすることである．すなわち，ゴールの実行を途中で一時中断 (suspend) したり，再開 (resume) したり，またその仕事を放棄 (abort) したりしたい．そのために考えられているのが，以下の形式の3引数 exec である[9]．

```
exec(G, I, O)
```

この exec のインプット・ストリーム I からは susp, resume, abort などのコントロール情報をいれることができる．たとえば，この exec の実行中にゴールの実行を一時中断したくなったら I を [susp|I′] と具体化する．このとき exec は休止状態になり，exec のインプット・ストリーム I′ だけが次の指令を待つ．exec を再開したくなったときには I′ を [resume|I″] と具体化する．exec 自体が不要となったときには，インプット・ストリーム I を [abort|I′] と具体化する．

またアウトプット・ストリームは，ゴール実行が成功すると [success]，ゴール実行が失敗すると [failure] に具体化される．

この3引数 exec は，以下のように記述できる．

```
exec(true, I, O) :- true|O = [success].
exec(false, I, O) :- true|O = [failure].
exec((A, B), I, O) :- ture|exec(A, I, O1),
        exec(B, I, O2), merge(O1, O2, O).
exec(A, I, O) :- not-sys(A), var(I)|
        reduce(A, I, Body, O, NewO),
        exec(Body, I, NewO).
exec(A, I, O) :- sys(A), var(I)|
        sys-exe(A, I, O).
exec(A, [susp|I], O) :- true|wait(A, I, O).
exec(A, [abort|I], O) :- true|O = [aborted].
```

```
wait(A, [resume|I], O) :- true|exec(A, I, O).
wait(A, [abort|I], O) :- true|O = [aborted].
```

exec は実行されるべきゴールが true であれば [success]，false であれば [failure] と O を具体化する．実行されるべきゴールが複数個あれば，それを分解し，一つ一つを実行する．またゴールがシステム定義述語でない（すなわちユーザ定義述語である）ときには，reduce でゴールを展開する．ゴールがシステム定義述語であれば，sys-exe でそれを解く．ここで var(I) は，exec のインプット・ストリームにメッセージが来ていないことを確かめるのに使われている．また，インプット・ストリームに（susp, resume, abort などの）コントロール情報が来ているときにはそれぞれ適切な処理を行なう．

この exec の特徴は，常に引数でメタレベルとの連絡がストリームとして確保されているということである．すなわち，ゴールの成功や失敗という概念は，絶対的な概念ではなくメタレベルへのメッセージと考えている．このメタインタプリタは，前の1引数 exec のメタインタプリタと比較した場合，メタレベルとオブジェクトレベルを，メタレベルからの制御やゴール実行の成功や失敗に関して明確に区別している．これは，（正確には対応しないが）メタレベルとオブジェクトレベルの区別という点で，Lisp でいえば 2-Lisp の段階に相当すると思われる．

次にこの GHC のメタインタプリタにスケジューリングキューとリダクションカウンタを陽に導入する．スケジューリングキューとリダクションカウンタを陽に導入した exec は，exec(H, T, I, O, MaxRC, RC) の6引数になる．この exec(H, T, I, O, MaxRC, RC) において，最初の二つの引数HとTはスケジューリングキューの頭部と尾部，次の二つの引数IとOはメタレベルとの入出力，MacRC はこの exec に許されたリダクションの数，RC は exec 実行時のリダクションの数である．

なお，ここで使われているHとTの差分リストによるスケジューリングキューの実現は，Prolog による Concurrent Prolog インタプリタの実現[10)]で使われた方法をまねたものである．頭部「H」と尾部「T」は，最初，スケジューリングキューが空のときには同じ変数を指しているが，Tにはゴールが入力され，Hではゴールが取り出されていく．HとTの差の部分がちょうどスケジューリ

ングキューに格納されているゴール列に相当する．（この技術を使用することにより，リストの結合などの操作が不要となる．）

6引数のメタインタプリタは以下のようになる．

```
exec(H, T, I, O, MaxRC, RC) :- true|
        O = [success(count = RC)].
exec([true|H], T, I, O, MaxRC, RC) :- true|
        exec(H, T, I, O, MaxRC, RC).
exec([false|H], T, I, O, MaxRC, RC) :- true|
        O = [failure(false)].
exec([A|H], T, I, O, MaxRC, RC) :-
        not-sys(A), var(I), MaxRC>RC|
        reduce(A, T, NT, O, NO, RC, RC1),
        exec(H, NT, I, NO, MaxRC, RC1).
exec([A|H], T, I, O, MaxRC, RC) :-
        sys(A), var(I), MaxRC>RC|
        sys-exe(A, T, NT, O, NO, RC, RC1),
        exec(H, NT, I, NO, MaxRC, RC1).
exec([A|H], T, I, O, MaxRC, RC) :-
        MaxRC ≦ RC|O = [count-over].
exec(H, T, [Mes|I], O, MaxRC, RC) :- true|
        control-exec(Mes, H, T, I, O, MaxRC, RC).
```

ここでは exec はスケジューリングキューのためにゴールを逐次的に処理する．今までの exec では，reduce がゴールを複数個のゴールに展開したときにはそれを分解して並列に処理していたのが，この exec ではそれをスケジューリングキューに入れて逐次的に処理していく．スケジューリングキューの導入は，プログラムに逐次性を導入するとともに，リダクションカウントなどのプログラムの実行管理を容易にする．（ところでここで導入される逐次性は，ゴールのスケジューリングに対してであり，システム全体が逐次的に実行されるとは限らないことに注意されたい．）

reduce は第1引数のゴールを評価し，それが展開可能であれば，展開したゴ

ールをスケジューリングキューの尾部に詰め，リダクションカウントを1だけ増加させる．ゴールの評価がサスペンドするときには，もとのゴールをそのままスケジューリングキューの尾部に詰め，リダクションカウントは進めない．sys-exe の場合も同様の処理を行なう．(ただしこの場合は，評価されるものがシステム述語であり，ゴールがそれ以上展開されることはない.) またこの exec のインプット・ストリームにメッセージが来ているときには control-exec でそれを解く．

ところでこの6引数 exec のメタインタプリタは，Lisp でいえば 3-Lisp のメタインタプリタに相当すると思われる．3-Lisp ではメタインタプリタの中で「環境」と「継続」が陽に表現されたが，一般に何を陽に表現するかはオブジェクトシステムの何を制御したいかによる．GHC では，スケジューリングキューは「継続」の代わりをすると考え，リダクションカウンタは CPU リソースの代わりをすると考えたわけである．(変数の束縛などの「環境」を扱いたい場合には，変数管理を陽に扱うようなメタインタプリタを書く必要があるが，そのようなメタインタプリタは複雑になる.)

3-Lisp の場合には，メタインタプリタの記述にリフレクティブな関数が必要で，それがリフレクティブタワーの問題を引き起こした．しかしながら，この GHC の場合にはメタインタプリタの記述にリフレクティブな述語は使われていないのでリフレクティブタワーの問題は生じない．(GHC の言語要素をもっと絞り込めば，3-Lisp と同じ問題が生ずる可能性もある．しかしながら，現在の我々の関心はこうした「メタのメタ」といった問題よりも，さまざまなリフレクティブな機能の検討や，それによるオペレーティング・システム的機能の実現にある.)

9.5.4　リフレクティブ述語のインプリメント

さて次の課題は，この exec の上でリフレクティブな述語をインプリメントすることである．そこでどのような述語をインプリメントするかが問題となる．3-Lisp の場合を振り返ってみると，まずメタインタプリタの中で「環境」と「継続」とを陽に表現し，通常の関数定義のほかにリフレクティブな関数の定義を許すようにし，その中では「環境」と「継続」とを自由にいじれたのであ

った.

GHC の場合も同様に,「リフレクティブ述語の定義」というのを考えるのも一つの可能性である. しかしながらここでは多少違った「直接的」な手法を採用する. すなわち, 我々がメタインタプリタの中で陽に表現したスケジューリングキューとリダクションカウンタについて, 以下の述語を用意する.

get-q(NH, NT) … exec のスケジューリングキューを差分リストの形で取り出し, その頭部と尾部を NH, NT に入れる.

put-q(NH, NT) … exec のスケジューリングキューを NH, NT に再設定する.

get-rc(MaxRC, RC) … exec から MaxRC と RC を取り出す.

put-rc(Max) … exec の最大リダクションカウントを Max に再設定する.

これらの述語をプログラムの中で使うことにより, ユーザは自由にスケジューリングキューやリダクションカウンタをいじれることになる.

次にこれらの述語をいかにインプリメントするかが問題となるが, 一般にリフレクティブな述語は自己の状態をいじる述語であるので, できるだけ手早く実行される必要がある. しかしながら6引数 exec では, ゴール実行の優先度という概念がない. そこでまず, それらの述語を手早く実行するための準備として, 急行スケジューリングキューを持つ exec を以下のように定義する.

```
exec(I, O, [G@exp|H], T, RC, MaxRC) :- var(I) |
        exec([G|ET], ET, I, O, H, T, RC, MaxRC).
exec(EH, EH, In, Out, H, T, RC, MaxRC) :- var(I) |
        exec(I, O, T, RC, MaxRC).
```

通常の6引数 exec は, プログラムの実行中に @exp の付いたゴール (急行ゴール) に出会ったら8引数 exec を呼び出す. 8引数 exec は通常のスケジューリングキューのほかに急行スケジューリングキューを持っており, 急行スケジューリングキューの中のゴールをまず実行する. 急行ゴールは展開されてもその結果が再び急行スケジューリングキューに入れられる. すなわち, 急行ゴールはほかのゴールに優先して処理が成される. そして急行スケジューリングキューが空になったら, 8引数 exec は再び6引数 exec に戻る.

この8引数 exec を用いて, たとえば get-q, put-q は以下のように定義できる.

```
exec([get-q(NH, NT)|EH], ET, I, O, H, T, RC, MaxRC) :- true|
        RC1 := RC + 1,
        NH = H,
        NT = T,
        exec(EH, ET, I, O, H, T, RC1, MaxRC).
exec([put-q(NH, NT)|EH], ET, I, O, H, T, RC, MaxRC) :- true|
        RC1 := RC + 1,
        exec(EH, ET, I, O, NH, NT, RC1, MaxRC).
```

このプログラムでわかるように，execはこれらのリフレクティブな述語に出会うと現在のexecの内部状態を与え，あるいは内部状態を指定された値に変更する．なお，ここではget-rcとput-rcのプログラムは省略したが，これらのプログラムも同様である．

次にこれらのリフレクティブな述語の使用例として，「プログラムの現在のリダクション数が，制限まであと100以内となったとき，ユーザが入力した数だけ制限を上に延ばすプログラム」を記述してみる．

```
check-rc :- true|
        get-rc(MaxRC, RC),
        RestRC := MaxRC - RC,
        check(MaxRC, RestRC).
check(MaxRC, RestRC) :- 100 ≧ RestRC|
        get-q(H, T),
        input(AddRC),
        NRC := MaxRC + AddRC,
        put-rc(NRC),
        T = [check-rc@exp|NT],
        put-q(H, NT).
check(MaxRC, RestRC) :- 100 < RestRC|
        get-q(H, T),
        T = [check-rc@exp|NT],
        put-q(H, NT).
```

check-rc に @exp を付け，ほかの実際に動かしたいユーザゴールとともに動かすことにより，上に述べた効果が得られる．

この例ではリフレクティブな述語が，メタレベルの状態を自由に取り出したり変更したりしている．しかしながら，これらはあまりにも直接的な「危険」な方法である．（こうした制御は，現実のオペレーティング・システムでは重要な機能である．）現在のところ我々の関心は「リフレクションの機能と検討とそれによるオペレーティング・システム的機能の実現」にあるのでこの方式を採用したが，ユーザにリフレクション機能を解放する方式としてどのようなものがよいかは今後の研究課題であろう．

9.6 ま と め

本章で解説したメタ計算やリフレクションといった研究課題は，いずれも最近脚光をあびて来た研究分野である．それまでも FOL などの研究はあったが，計算機構としてのリフレクションの概念について，直接の発端は1984年の POPL での B.C. Smith の 3-Lisp の発表である．その後リフレクションに関してさまざまな研究が行なわれるようになり，1986年10月には，メタレベルアーキテクチャとリフレクションに関するワークショップがイタリアの Alghero で開催されている．

この分野の参考文献としては，イタリアの Alghero でのワークショップの論文をまとめ本にした文献11）が適切であろう．また，論理型言語に興味を持つ読者には，Meta-programming in Logic というワークショップが1988年6月と1990年4月に開かれているのでそのプロシーディングスを読まれることを勧める．最近日本でもリフレクションに関して興味を持つ研究者が増えてきた．今後，これらの分野における研究の展開には目の離せない状況となってきている．

参考文献

1) Maes, P.: Computational Reflection, Ph.D. Thesis, Laboratory for Artificial Intelligence, Vrije Unversiteit Brussel, Belgium, Jan, 1987.
2) Watanabe, T.: Reflection in Object-Oriented Concurrent Systems, M.S. Thesis,

Tokyo Institute of Technology, Feb. 1988.

3) McCarthy, J. et al.: *LISP 1.5 Programmer's Manual*, Cambridge, Mass., The MIT Press, 1965.

4) Steele, G. and Sussman, G.: The Art of Interpreter, or The Modularity Complex (Parts Zero, One, and Two), M.I.T. Artificial Intelligence Laboratory Memo AIM-453, 1978.

5) Bowen, K. and Kowalski, R.: *Amalgamating language and metalanguage in logic programming*, Logic Programming, pp. 153–172, Academic Press, London, 1982.

6) Weyhrach, R.: Prolegomena to a Theory of Mechanized Formal Reasoning, In *Artificial Intelligence*, Vol. 13, No. 1, 2, pp. 133–170, 1980.

7) Smith, B.C.: Reflection and Semantics in Lisp, In. *Proc. of the POPL*, Salt Lake City, Utah, pp. 23–35, 1984.

8) 古川康一・溝口文雄共編：並列論理型言語 GHC とその応用，共立出版，1987.

9) Clark, K. and Gregory, S.: Notes on Systems Programming in Parlog, In *Proc. of FGCS'84*, pp. 299–306, *ICOT*, 1984.

10) Shapiro, E.: A Subset of Concurrent Prolog and Its Interpreter, *Concurrent Prolog: Collected Papers*, Vol. 1, pp. 27–83, The MIT Press, 1987.

11) Maes, P. and Nardi, D. eds.: *Meta-Level Architectures and Reflection*, North-Holland, 1988.

（たなか じろう）

索　引

〔サ　行〕

〔タ　行〕

〔ナ　行〕

〔ハ　行〕

〔D・E・F〕

〔G・H・I〕

〔K・L・M〕

〔N・O・P・Q〕

〔R・T・U〕

編者略歴

井田哲雄（いだ てつお）

1971年　東京大学教養学部基礎科学科卒業
1975年　東京大学大学院理学系研究科博士課程（物理学専攻）中退
　　　　理化学研究所情報科学研究室研究員
現　在　筑波大学電子・情報工学系助教授，理学博士

田中二郎（たなか じろう）

1975年　東京大学理学部物理学科卒業
1984年　米国ユタ大学計算機科学科博士課程卒業，Ph. D.
1985–1988年　(財)新世代コンピュータ技術開発機構(ICOT)，主任研究員
現　在　富士通(株)国際情報社会科学研究所に勤務

続　新しいプログラミング・パラダイム

検印廃止

1990年11月25日　初版1刷発行

編　者　井田哲雄
　　　　田中二郎

発行者　南條正男
東京都文京区小日向4丁目6番19号

印刷者　竹内一
東京都荒川区東日暮里6丁目20番9号

NDC 007.64

発行所　東京都文京区小日向4丁目6番19号
電話　東京 947 局 2511 番（代表）
郵便番号112　振替口座　東京 1-57035番
共立出版株式会社

印刷・壮光舎印刷　製本・中條製本　Printed in Japan

社団法人
自然科学書協会
会員

ISBN 4-320-02535-0